ATLANTIS PRESS

Atlantis Studies in Computing
Series Editors: Jan A. Bergstra · Michael W. Mislove

Andrea Marino

Analysis and Enumeration

Algorithms for Biological Graphs

Atlantis Studies in Computing

Volume 6

Aims and Scope of the Series

The series aims at publishing books in the areas of computer science, computer and network technology, IT management, information technology and informatics from the technological, managerial, theoretical/fundamental, social or historical perspective.

We welcome books in the following categories:

Technical monographs: these will be reviewed as to timeliness, usefulness, relevance, completeness and clarity of presentation.
Textbooks.

Books of a more speculative nature: these will be reviewed as to relevance and clarity of presentation.

For more information on this series and our other book series, please visit our website at:

www.atlantis-press.com/publications/books
Atlantis Press
29, avenue Laumière
75019 Paris, France

More information about this series at http://www.springer.com/series/10530

Andrea Marino

Analysis and Enumeration

Algorithms for Biological Graphs

Andrea Marino
Dipartimento di Informatica
Milan
Italy

ISSN 2212-8557 ISSN 2212-8565 (electronic)
Atlantis Studies in Computing
ISBN 978-94-6239-096-6 ISBN 978-94-6239-097-3 (eBook)
DOI 10.2991/978-94-6239-097-3

Library of Congress Control Number: 2015933151

To
My Parents, Maria, Giovanna,
Marco, and Alessandro, Lucilla.

Foreword 1

The Italian Chapter of the EATCS (European Association for Theoretical Computer Science) was founded in 1988, and aims at facilitating the exchange of ideas and results among Italian theoretical computer scientists, and at stimulating cooperation between the theoretical and the applied communities in Italy.

One of the major activities of this Chapter is to promote research in theoretical computer science, stimulating scientific excellence by supporting and encouraging the very best and creative young Italian theoretical computer scientists. This is done also by sponsoring a prize for the best Ph.D. thesis. An interdisciplinary committee selects the best two Ph.D. theses, among those defended in the previous year, one on the themes of Algorithms, Automata, Complexity and Game Theory and the other on the themes of Logics, Semantics and Programming Theory.

In 2012 we started a cooperation with Atlantis Press so that the selected Ph.D. theses would be published as volumes in the Atlantis Studies in Computing.

The present volume contains one of the two theses selected for publication in 2014:

Type Disciplines for Systems Biology by Livio Bioglio (supervisor: Prof. Mariangiola Dezani, University of Torino, Italy)

and

Algorithms for Biological Graphs: *Analysis and Enumeration* by Andrea Marino (supervisor: Prof. Pierluigi Crescenzi, University of Firenze, Italy).

The scientific committee which selected these theses was composed of Profs. Franco Barbanera (University of Catania), Arturo Carpi (University of Perugia) and Rossella Petreschi (*Sapienza* University of Rome).

They gave the following reasons to justify the assignment of the award to the thesis by Andrea Marino:

The Ph.D. dissertation "Algorithms for biological graphs: analysis and enumeration" *by Andrea Marino deals with efficient algorithms for enumeration problems on graphs. The main application fields for these algorithms are biological and social networks, for which data can be conveniently modeled as graphs. This thesis presents both deep theoretical results and extensive experimental implementations.*

Moreover, in Chap. 2, an overview of basic techniques used for enumeration algorithms is reported. Namely in this thesis it is possible to find algorithms for enumerating:

- *all diametral and radial vertices;*
- *all maximal directed acyclic sub-graphs of which sources and targets belong to a predefined subset of the vertices (stories);*
- *all cycles and/or paths in an undirected graph;*
- *all pairs of (s, t)-paths sharing only nodes s and t ((s, t)-bubbles).*

Summarizing, this thesis contains several important contributions in the area of graph algorithms and can be considered an important reference for all the researchers that have to work with enumerating problems.

I would like to thank the members of the scientific committee, and I hope that this initiative will further contribute to strengthen the sense of belonging to the same community of all the young researchers that have accepted the challenges posed by any branch of theoretical computer science.

Rome, January 2015

Tiziana Calamoneri
President of the Italian Chapter of EATCS

Foreword 2

The development of algorithms for enumerating all possible solutions of a specific combinatorial problem has a long history, which dates back to, at least, the 1960s, when the problem of enumerating some specific graph-theoretic structures (such shortest paths and cycles) has been attacked. As already observed by David Eppstein in 1997, these enumeration problems have several applications, such as (1) looking for structures which satisfy some additional constraints which are hard to optimize, (2) evaluating the quality of a model for a specific problem, in terms of the number of incorrect structures, (3) computing how sensitive the structures are to variation of some problem's parameters and (4) examining not just the optimal structures, but a larger class of structures, to gain a better understanding of the problem. As a matter of fact, in the last 50 years a large variety of enumeration problems have been considered in the literature, ranging from geometry problems to graph and hypergraph problems, from order and permutation problems to logic problems, and from set problems to string problems. A very recent compendium has been compiled by Kunihiro Wasa, which includes 350 combinatorial problems and more than 230 references. Nevertheless, the research area of enumeration algorithms is still very active and still includes many interesting open problems. This is where this book comes into play, by first presenting an overview of the main computational issues related to the design and analysis of enumeration algorithms, and by then contributing to this research area with several significant results, both theoretical and experimental.

Although the emphasis of the book is on enumeration problems, it is worth noting that the original main application area of the thesis of Andrea Marino has been computational biology. Indeed, in the previous years, biologists have accumulated a huge amount of information, at different levels of observation, from the molecular level to the population one. This information usually describes interactions or relationships among entities of biological nature, and they are often represented by means of networks (or, equivalently, graphs). Graphs allow researchers to abstract from the specific individual information: the complexity of a biological entity is enclosed into a vertex of the network and the complex interaction

mechanisms between two entities are simply described by means of an arc. Clearly, the biological application determines the meaning of the nodes and of the arcs and influences the network topology: typical networks at the molecular level are gene regulation networks, protein interaction networks and metabolic networks, while typical networks at the macroscopic level are, instead, phylogenetic networks and ecological networks. Reducing problems arising in biology to the analysis of networks allows us to take advantage of the many results and algorithmic techniques that have been developed in graph theory and, more recently, in the analysis of complex networks. In other words, the observation of biological phenomena is turned into the observation of the network, of its structure and of its properties. The network becomes a tool to investigate the macromolecular interactions at the level of genes, metabolites and proteins to extract the cellular phenotypes, or the conglomerate of several cellular processes resulting from the expression of the genes and of the proteins.

The main goal of this book is the application of algorithm design and complexity analysis techniques to the analysis of biological (and, more in general, of complex) networks, by focusing mainly on topological property computation and subnetwork extraction tasks. Several quantifiable tools of network theory offer unforeseen possibilities to understand biological network organization and evolution. Some well-known examples of these tools are measures like the degree distribution, the diameter (that is, the longest shortest path) and the clustering coefficient. These topological properties of biological networks can be seen as the result of a network evolution process: hence, one can formulate evolving network models for biological networks which produce networks consistent with the above topological properties. This implies that efficient algorithms have to be designed in order to compute these properties in a very little amount of time and (maybe more importantly) of space (note that, sometimes, even polynomial-time/space algorithms might turn out to be too expensive if a massive experimentation has to be done and/or if the size of the network is quite large). For what concerns the second task, that is, subnetwork extraction, observe that, in general terms, this task consists in extracting a subgraph that best explains the relationships between a given set of nodes of interest in a graph. A typical example in communication networks of such a problem is the Steiner tree problem which consists in finding the lightest tree connecting a specific subset of vertices of the network. Subnetwork extraction is a common tool while studying biological networks: for example, in 2010, Faust et al. investigated six different approaches, all based on subnetwork extraction, to extract relevant pathways from metabolic networks. One of the main issues with the subgraph extraction approach is to determine the kind of subgraph to be extracted, which clearly has to be meaningful from a biological point of view. After that, even in this case it turns out that most of the times the extraction of desired subgraphs is a computationally difficult problem. Finally, as it is common in the bioinformatics research area, finding one subgraph is not usually enough: no clear optimization criterium is usually known, so that the problem becomes even more difficult since it requires to enumerate all possible subgraphs.

All the enumeration problems attacked in this book arise from real-world applications, either in the specific field of computational biology or in the more general field of complex networks. Some of these problems (such as the enumeration of cycles and the enumeration of diametral vertices) were already well known and widely studied. Others (such as the enumeration of bubbles and the enumeration of stories) are closely related to previously known problems (such as the enumeration of cycles and the enumeration of feedback vertex sets). For all these problems, efficient algorithms and/or heuristics are proposed in order to deal with them: all these algorithms have been implemented and experimented, thus validating their usefulness in solving the original application problem. Indeed, one of the main beautiful characteristics of this book is the fact that it combines deep theoretical results and practical implementation and experimentation: thus significantly contributing both to the solution of (biological) very interesting practical questions and to the field of theoretical computer science. In particular, I would like to emphasize one of the most impressive theoretical results contained in this book, that is, the first optimal algorithm for enumerating cycles in an undirected graph. This result significantly improves the solution of a 40-year-old problem! And I would also like to emphasize one of the most impressive experimental results contained in this book: that is, the design, analysis and implementation of a new very efficient heuristic for enumerating diametral vertices in a graph. By making use of these new heuristics, for example, the diameter of a snapshot of a subgraph of the Facebook network, that contained approximately 150 millions of vertices and almost 16 billions of edges, has been computed in just 20 minutes (for the sake of curiosity, the diameter value is 41)!

In summary, I think that this book is a very cute example of how theory and practice should proceed together, by exploiting the "virtuous circle" in which practical problems (in this case, mostly biological ones) motivate significant and deep contributions to theoretical computer science, which in turn allow efficient, useful and practical solutions to the original problems.

Florence, January 2015 Pierluigi Crescenzi

Acknowledgments

Thanks to my Ph.D. supervisor Pierluigi Crescenzi: this work would not have been possible without his guidance.

Thanks to Marie-France Sagot and BAMBOO & BAOBAB Team of INRIA Grenoble Rhône-Alpes in Lyon, that supported a big part of this work.

Thanks to Roberto Grossi, Michel Habib and LIAFA of University Paris Diderot, Paris 7.

Thanks to Fedor Fomin and Andrea Pietracaprina, who kindly accepted to read and review the thesis on which this work is based, for their helpful comments and suggestions.

Thanks to the Italian Chapter of the EATCS (European Association for Theoretical Computer Science) for the Best Italian Ph.D. Thesis Award in Theoretical Computer Science 2013 for the track "Algorithms, Automata, Complexity and Game Theory", which gave to me the opportunity to write this book.

This book is the result of a joint work with: Vicente Acuña, Etienne Birmelé, Matteo Brilli, Ludovic Cottret, Pierluigi Crescenzi, Rui A. Ferreira, Roberto Grossi, Michel Habib, Cecilia Klein, Vincent Lacroix, Leonardo Lanzi, Alberto Marchetti-Spaccamela, Paulo Vieira Milreu, Nadia Pisanti, Romeo Rizzi, Gustavo Akio Tominaga Sacomoto, Marie-France Sagot, Leen Stougie and Takeaki Uno. Thanks to all my coauthors.

Contents

Chapter 1
Introduction

The aim of enumeration is listing all the feasible solutions of a given problem: this is particularly useful whenever the goal of the problem is not clear and we need to check all its solutions.

Since the number of solutions to be enumerated is often exponential with respect to the size of the input, enumeration algorithms require often at least exponential time. Whenever the size of the input is small, brute force algorithms are helpful: in this case the algorithm produces the solutions one after the other by checking if the current solution has been already generated or not. However, when the number of solutions grows up the time needed to produce a new solution heavily increases. In such a context, the complexity classes of enumeration problems are defined depending on the number of solutions, so that if the number of solutions is small, an efficient algorithm has to terminate after short (polynomial) time, otherwise it is allowed to spend more time. According to this idea in 1988 in a popular paper by Johnson, Papadimitriou, and Yannakakis the main complexity classes have been defined [1]: "the least that we could ask is that the time required to output all solutions be bounded by a polynomial in n (*the size of the input*) and C (*the number of solutions*)" (Polynomial Total Time), while more strictly we could require that "the delay between any two consecutive solutions is bounded by a polynomial in the input size" (Polynomial Delay). In other words, while the first imposes a polynomial average delay between two consecutive solutions, the second class imposes a fixed polynomial delay between any two consecutive solutions.

In this work, we will show some new examples of enumeration algorithms that (in some sense) can fit with the above categories, in particular we will show how to: enumerate *stories* by making use of an efficient brute force algorithm; enumerate *bubbles* by using a polynomial (linear) delay algorithm; enumerate *paths or cycles* by using an optimal algorithm whose total time is bounded by the size of the paths or cycles to be enumerated. Moreover, in the last part we will talk also about enumerating *central and peripheral* vertices of a network (and computing diameter and radius): the number of solutions of this latter problem is polynomial, but since it is often applied to real world huge networks, a linear time algorithm is desirable.

A. Marino, *Analysis and Enumeration*, Atlantis Studies in Computing 6,
DOI 10.2991/978-94-6239-097-3_1

All the problems above were motivated by some biological problems modelled by using biological networks. However, even if all the problems we will discuss share the common biological application, the corresponding computational problems we define are of more general interest and our results hold in the case of arbitrary networks. In the following we will briefly introduce the application and explain what *stories*, *bubbles*, *paths*, *cycles*, *central*, and *peripheral* vertices are. We will then overview our results, giving the main references to find the original works. We will use the standard notations and definitions, described in Sect. 1.6.

1.1 An Application: Biological Graph Analysis

Since one peculiar property of biological networks is the uncertainty, a scenario in which enumeration algorithms can be helpful is biological network analysis. Modelling biological networks indeed introduces bias: arc dependencies are neglected and underlying hyper-graph behaviours are forced in simple graph representations to avoid intractability. Moreover, the dynamical behaviours of biological networks are often not considered: indeed most of the currently available biological network reconstructions are potential networks, where all the possible connections are indicated, even if edges/arcs and vertices are hardly present all together at the same time. For these aspects of biological networks, we invite the reader to see the following work.

[2] Cecilia Klein, Andrea Marino, Marie-France Sagot, Paulo Vieira Milreu, and Matteo Brilli. Structural and dynamical analysis of biological networks. *Briefings in functional genomics*, 2012.

In this scenario, when defining and solving a problem on a biological network, it is quite natural for a biologist to ask all the solutions to check whether these make sense or they are merely artifacts of the model.

1.2 Enumerating Stories

The problem of enumerating stories was motivated initially by the biological question in [3] related to Metabolic networks, in particular to compound graphs, in which vertices are compounds and there is an arc from a compound x to a compound y if there is a metabolic reaction that consumes x and produces y (see Sect. 3.2.2). A subset $\mathbb{B}$ corresponds to compounds that have been experimentally identified as having a significantly higher or lower production in a given condition (for instance when an organism is exposed to some stress). The aim is then to extract all the interaction dependencies among the compounds in $\mathbb{B}$ which do not create cycles but at the same time involve as many compounds as possible. These may require intermediate steps that concern compounds not in $\mathbb{B}$, but the initial and final steps

must involve only compounds in $\mathbb{B}$. A solution, that is a possible scenario of metabolic dependencies, is called a *(metabolic) story*.

A metabolic story has to capture the relationship between the vertices of interest in a way that allows us to define a flow of matter from a set of sources to a set of target compounds. The need for this hierarchy between the compounds led us to consider acyclic solutions. The maximality condition has been added in order to capture all alternative paths between the sources and the targets. The problem is then to "tell" all possible stories given as input a graph G and a subset $\mathbb{B}$ of the vertices of G.

We will present a polynomial algorithm to find one story and an exact but exponential approach for the enumeration problem [4, 5]. This definition is a generalization of a well-known problem which is the *feedback arc set* problem. However, any polynomial-delay algorithm to enumerate feedback arc sets (for instance [6]) can only be used in some particular instances that, as we have shown in [4, 5], correspond to graphs encoding a Metabolic network which do not contain the so-called "bad vertices", which are any not interesting vertex v such that for any predecessor p of v and for any successor s of v, there exists a cycle containing the arcs (p, v) and (v, s). Moreover we will show that finding a story with a specified set of sources or targets is NP-hard.

Our contribution appeared in the following works.

[4] Vicente Acuña, Etienne Birmelé, Ludovic Cottret, Pierluigi Crescenzi, Vincent Lacroix, Alberto Marchetti-Spaccamela, Andrea Marino, Paulo Vieira Milreu, Marie-France Sagot, and Leen Stougie. Telling stories. Workshop on Graph Algorithms and Applications selected for submission to the special issue of Theoretical Computer Science in honor of Giorgio Ausiello in the occasion of his 70th birthday, 2011.

[5] Vicente Acuña, Etienne Birmelé, Ludovic Cottret, Pierluigi Crescenzi, Fabien Jourdan, Vincent Lacroix, Alberto Marchetti-Spaccamela, Andrea Marino, Paulo Vieira Milreu, Marie-France Sagot, and Leen Stougie. Telling stories: Enumerating maximal directed acyclic graphs with a constrained set of sources and targets. *Theor. Comput. Sci.*, 457:1–9, 2012.

The open problems arising from these works have been presented in the following workshop.

[7] Vicente Acuña, Etienne Birmelé, Ludovic Cottret, Pierluigi Crescenzi, Fabien Jourdan, Vincent Lacroix, Alberto Marchetti-Spaccamela, Andrea Marino, Paulo V. Milreu, Marie-France Sagot, and Leen Stougie. Metabolic stories: uncovering all possible scenarios for interpreting metabolomics data. *In First RECOMB Satellite Conference on Open Problems in Algorithmic Biology (RECOMB-AB)*, 2012.

1.3 Enumerating Bubbles

A DNA fragment, that is an RNA-coding sequence, is transformed in a Pre-mRNA sequence, through the transcription phase, in which sequences of *exons* and sequences of *introns* alternatively occur. The removal of all the sequences of introns and of some sequences of exons leads to the mRNA sequence that is a protein-coding sequence that translated leads to a protein. Since not any exon is transcribed in the mRNA sequence, there can be many possible mRNA sequences. For instance, let $\langle e_1, i_1, e_2, i_2, e_3, i_3, e_4, i_4\rangle$ be a fragment of DNA, where for any j, with $1 \leq j \leq 3$, e_j and i_j are the jth sequence of exons and introns respectively. The possible resulting mRNA sequences containing e_1 are $\langle e_1, e_2, e_3, e_4\rangle$, $\langle e_1, e_2, e_3\rangle$, $\langle e_1, e_2, e_4\rangle$, $\langle e_1, e_3, e_4\rangle$, $\langle e_1, e_2\rangle$, $\langle e_1, e_3\rangle$, $\langle e_1, e_4\rangle$. The underlying phenomenon is called alternative splicing and checking all the alternative events has been shown in [8] to correspond to checking recognisable patterns in a de Bruijn graph built from the reads provided by a sequencing project (see Sect. 3.2.4). The pattern corresponds to an (s, t)-bubble: an (s, t)-bubble is a pair of vertex-disjoint (s, t)-paths that only shares s and t.

Since the k-mers correspond to all words of length k present in the reads (strings) of the input dataset, and only those, in relation to the classical de Bruijn graph for all possible words of size k, the de Bruijn graph for NGS data may then not be complete. We will ignore all the details related to the treatment of NGS data using De Bruijn graphs, and consider instead the more general case of finding all (s, t)-bubbles in an arbitrary directed graph.

In particular we show the first linear delay algorithm to identify all bubbles. A previous known algorithm presented in [8] was an adaptation of Tiernan's algorithm for cycle enumeration [9] that does not have a polynomial delay. In the worst case the time elapsed between the outputs of two solutions is proportional to the number of paths in the graph, i.e. exponential in the size of the graph. Our algorithm is a non-trivial adaptation of Johnson's cycle enumeration algorithm [10] in a directed graph with the same theoretical complexity. Notably, the method we propose enumerates all bubbles with a given source with $O(|V| + |E|)$ delay. The algorithm requires an initial transformation of the graph, for each source s, that takes $O(|V|+|E|)$ time and space; this transformation reduces the enumeration of bubbles to the enumeration of constrained cycles in a special graph.

Our algorithm is the result of the following work.

[11] Etienne Birmelé, Pierluigi Crescenzi, Rui A. Ferreira, Roberto Grossi, Vincent Lacroix, Andrea Marino, Nadia Pisanti, Gustavo Akio Tominaga Sacomoto, and Marie-France Sagot. Efficient bubble enumeration in directed graphs. In *String Processing and Information Retrieval - 19th International Symposium, SPIRE 2012*, pages 118–129, 2012.

1.4 Enumerating Cycles or Paths

Studying paths or cycles of biological networks can be useful for several purposes. In the case of interaction graphs, such as Gene Regulatory networks, the importance of enumeration has been shown in [12]. These networks are directed, their vertices are genes, and their arcs are signed, where the sign or weight of the arcs indicates the causal relationship between the vertices, such as activation or inhibition (see Sect. 3.2.3). In particular, cycles and paths can be useful for studying dependencies among vertices, the steady state and multistationarity of dynamic models. Moreover cycles and paths respectively correspond to feedback loops [13, 14] related to robustness in cell signaling networks [15], and signaling paths, i.e. the different positive and negative routes along which a molecule can affect another.

We have considered the problem of enumerating paths and cycles in the case of undirected graphs. This result can be useful for undirected Protein-Protein Interaction networks, where vertices are proteins and edges are interactions (see Sect. 3.2.1), but in the case of interaction networks in general, our approach neglects the effects of the controls, i.e. the sign and direction of the arcs.

Listing all the paths and cycles in a graph is a classical problem whose efficient solutions date back to the early 70s. The best known solution in the literature is given by Johnson's algorithm [10] and takes $O((|\mathscr{C}(G)| + 1)(|E| + |V|))$ and $O((|\mathscr{P}_{st}(G)| + 1)(|E| + |V|))$ time for a graph $G = (V, E)$, where $\mathscr{C}(G)$ and $\mathscr{P}_{st}(G)$ denote respectively the set of cycles and (s, t)-paths in G. However there exists graphs for which this algorithm is not optimal.

We will present the first optimal algorithm to list all the paths and cycles in an undirected graph G. Our algorithm requires $O(|E| + \sum_{c \in \mathscr{C}(G)} |c|)$ time and is asymptotically optimal: indeed, $\Omega(|E|)$ time is necessarily required to read G as input, and $\Omega(\sum_{c \in \mathscr{C}(G)} |c|)$ time is necessarily required to list the output. Moreover, our algorithm lists all the (s, t)-paths in G optimally in $O(|E| + \sum_{\pi \in \mathscr{P}_{st}(G)} |\pi|)$ time, observing that $\Omega(\sum_{\pi \in \mathscr{P}_{st}(G)} |\pi|)$ time is necessarily required to list the output.

Our algorithm exploits the decomposition of the graph into biconnected components and without loss of generality restricts to study paths and cycles in a same biconnected component. Thus it recursively lists the cycles or (s, t)-paths using the classical binary partition: given an edge e in G, list all the solutions containing e, and then all the solutions not containing e, at each time modifying the graph. In order to avoid recursive calls (in the binary partition) that do not list solutions, we will use a *certificate*, as a data structure, whose cost for dynamically updating is constant with respect to the number of solutions produced. In order to prove the complexity obtained, we will exploit the properties of the binary recursion tree corresponding to the binary partition.

This work appeared in the following.

[16] Etienne Birmelé, Rui A. Ferreira, Roberto Grossi, Andrea Marino, Nadia Pisanti, Romeo Rizzi, and Gustavo Sacomoto. Optimal listing of cycles and st-paths in undirected graphs. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2013*, pages 1884–1896, 2013.

1.5 Further Analysis: Enumerating Central and Peripheral Vertices

The structural analysis of real world networks, such as citation, collaboration, communication, road, social, and web networks, has attracted a lot of attention and the fundamental analysis measures have been reviewed in [17]. An aim of structural analysis is the identification of important and not important vertices within a network. In the biological domain, the importance of a vertex can be defined in many different ways. With neighbourhood-based centrality measures, such as degree, the importance of the vertices is inferred from their local connectivity and the more connections a vertex has the more central it is. Closeness, eccentricity, and shortest path based betweenness rely on global properties of a network, such as distance between vertices.

We will focus on the enumeration of the radial and diametral vertices, i.e. vertices that are central and peripheral according to the eccentricity notion of centrality, and on the computation of the radius and diameter of real world graphs. The diameter and radius of a graph are respectively the maximum and minimum eccentricity among all its vertices, where the eccentricity of a vertex x is the distance from x to its farthest vertex.

Thus, intuitively, the diametral source vertices are the vertices that hardly reach the others, the diametral target vertices are the vertices hardly reachable from the other ones, and the radial vertices are the vertices that easily reach all the vertices of the network. In order to calculate the vertices that can be easily reached from any other vertex, it is sufficient to consider the transposed graph.

We will present the D*i*FUB Algorithm, which is able to list all the diametral sources and targets and to compute the diameter of (strongly) connected components of a graph $G = (V, E)$ in time $O(|E|)$ in practice, even if, in the worst case, the complexity is $\Theta(|V||E|)$. Analogously, we will present a new algorithm to list all the central vertices and to compute the radius of (strongly) connected components of a graph in *almost* $O(|E|)$ time in practice.

This running time allows to compute radius and diameter of real world networks in practice. Indeed, the size of these networks has been increasing rapidly, so that in order to study such measures, algorithms able to handle huge amount of data are needed. Since the algorithms available until now were not able to compute diameter and radius in the case of huge real world graphs, the contribution of our algorithms is not just limited to biological networks analysis, but extends also to the analysis of complex networks in general. We thus have shown their effectiveness also for several other kinds of complex networks.

Our work appeared in the following.

[18] Pierluigi Crescenzi, Roberto Grossi, Leonardo Lanzi, and Andrea Marino. On computing the diameter of real-world directed (weighted) graphs. In *Experimental Algorithms - 11th International Symposium, SEA 2012*, pages 99–110, 2012.

This has been the generalization of the following works.

[19] Pierluigi Crescenzi, Roberto Grossi, Claudio Imbrenda, Leonardo Lanzi, and Andrea Marino. Finding the diameter in real-world graphs - experimentally turning a lower bound into an upper bound. In *Algorithms - ESA 2010, 18th Annual European Symposium. Proceedings, Part I*, pages 302–313, 2010.

[20] Pierluigi Crescenzi, Roberto Grossi, Michel Habib, Leonardo Lanzi, and Andrea Marino. On Computing the Diameter of Real-World Undirected Graphs. Workshop on Graph Algorithms and Applications selected for submission to the special issue of Theoretical Computer Science in honor of Giorgio Ausiello in the occasion of his 70th birthday, 2011.

[21] Pilu Crescenzi, Roberto Grossi, Michel Habib, Leonardo Lanzi, and Andrea Marino. On computing the diameter of real-world undirected graphs. *Theor. Comput. Sci.*, 514:84–95, 2013.

Our algorithm in [21], has been used to compute the diameter of Facebook Network (721.1M vertices, 68.7G edges, and diameter 41) with just 17 BFSes in a popular work ([22, 23], divulged by New York Times on November 22, 2011).

1.6 Basic Definitions and Notations

Given a set $X = \{x_1, \ldots, x_n\}$, the cardinality of X is denoted by $|X|$. The power set 2^X is the set of all subsets (including the empty set) of X. A sequence S is an ordered set and is denoted by $\langle s_1, \ldots, s_n \rangle$. The length of the sequence is also denoted by $|S|$. The concatenation of S with an element s_{n+1} is the sequence $\langle s_1, \ldots, s_n, s_{n+1} \rangle$ and is denoted by $\langle S, s_{n+1} \rangle$.

A graph G is a pair of sets (V, E), where the elements of V are called vertices and the elements of E are pairs of vertices, so that $E \subseteq V \times V$. In the case of undirected graphs, these pairs are not ordered, so that $(x, y) \in E$ implies $(y, x) \in E$, and we refer to them as *edges*, while for directed graphs, these pairs are ordered and called *arcs*. In the following we will denote by $n = |V|$ the number of vertices and $m = |E|$ the number of edges or arcs. For any arc (x, y), we say that it is from x to y, or it is incoming to y and out-going from x, or x is the out-neighbour of y and y is the in-neighbour of x, or y is a successor of x and x is a predecessor of y. For any edge (x, y) we say that x and y are neighbours. Any edge or arc (x, x) is called self-loop.

If E is a multi-set, then G is called multi-graph, otherwise it is called simple graph. If not specified, we will refer to simple graphs simply as graphs.

For a vertex $u \in V$, for an undirected graph we denote by $N(u)$ its neighbourhood and by $d(u) = |N(u)|$ its degree, while for a directed graph we denote by $N^+(u)$ and $N^-(u)$ its out- and in-neighbourhood respectively, and by $d^+(u) = |N^+(u)|$ and $d^-(u) = |N^-(u)|$ its out- and in-degree respectively. Vertex u is called *source* if $d^+(u) = 0$ and $d^-(u) > 0$ and *target* if $d^-(u) = 0$ and $d^+(u) > 0$.

For a directed graph $G = (V, E)$, we define its transposed graph as $G' = (V, E')$, where $E' = \{(u, v) : (v, u) \in E\}$.

A path π is a sequence of vertices $\langle v_1, \ldots, v_k \rangle$, such that for any i with $1 < i \leq k$, v_i is neighbour or out-neighbour of v_{i-1}. Thus, we refer to a path π by its natural sequence of vertices or arcs/edges. A path π from s to t, or (s, t)-*path*, is denoted by $\pi = s \rightsquigarrow t$. Additionally, $\mathscr{P}(G)$ is the set of all paths in G and $\mathscr{P}_{s,t}(G)$ is the set of all (s, t)-paths in G. When $s = t$ we have cycles, and $\mathscr{C}(G)$ denotes the set of all cycles in G. If a directed graph does not contain cycles, then it is called Directed Acyclic graph (in short, DAG). Whenever for any pair of vertices u, v, there is a path from u to v, we say that the graph is connected if G is undirected, or strongly connected if G is directed.

The number of arcs or edges in a path π is called length and denoted by $|\pi|$. Analogously the number of arcs or edges in a cycle c is called length and denoted by $|c|$. In this work, we will consider just simple paths and simple cycles.

For any two vertices u, v, the length of the shortest path from u to v is called *distance* and denoted by $d(u, v)$, that is $d(u, v) = min_{\pi \in \mathscr{P}_{u,v}(G)} |\pi|$. Whenever there is no path from u to v, v is said to be not reachable from u and $d(u, v) = \infty$. The diameter of G is the minimum D such that for any pair of vertices $u, v, d(u, v)$ is less or equal than D, that is $D = \max_{u,v \in V \times V} d(u, v)$. We define the forward (respectively, backward) eccentricity of u and denote it by $ecc_F(u)$ (respectively, $ecc_B(u)$) the $\max_{v \in V} d(u, v)$ (respectively, $\max_{v \in V} d(v, u)$). In the case of undirected graphs, forward and backward eccentricities coincide and are both called simply eccentricity and denoted by $ecc(u)$. Thus, the diameter is defined as the maximum forward or the maximum backward eccentricity, i.e. $D = \max_{u \in V} ecc_F(u) = \max_{u \in V} ecc_B(u)$. The radius R of G is the minimum forward eccentricity of its vertices, i.e. $R = \min_{u \in V} ecc_F(u)$, or, in the case of undirected graphs, $R = \min_{u \in V} ecc(u)$. Notice that in general, in directed graphs $\min_{u \in V} ecc_F(u) \neq \min_{u \in V} ecc_B(u)$. We denote by T_u^F (respectively, T_u^B) a forward (respectively, backward) Breadth-First Search (in short, BFS) tree rooted at node u, so that $\mathrm{ecc}_F(u)$ (respectively, $\mathrm{ecc}_B(u)$) is its height. In an undirected graph for any vertex u in V, the levels of the forward breadth-first search tree rooted at node u, T_u^F, coincide with a backward BFS tree rooted at the same node, T_u^B: thus we refer to both trees simply by T_u.

For a vertex $v \in V$, the *postorder* DFS number of v is the relative time in which v was *last* visited in a Depth-First Search (in short, DFS) traversal, i.e. the position of v in the vertex list ordered by the last visiting time of each vertex in the DFS.

The subgraph *induced* by a set of vertices $V' \subseteq V$ is a graph $G' = (V', E')$, where $E' = \{(u, v) : (u, v) \in E, u, v \in V'\}$. Thus, $G[V']$ denotes the subgraph induced by V', and $G - u$ is the induced subgraph $G[V \setminus \{u\}]$ for $u \in V$. Likewise for $e \in E$, we adopt the notation $G - e = (V, E \setminus \{e\})$, and, for any $F \subseteq E$, $G - F = (V, E \setminus F)$.

A *rooted tree* T is an undirected graph such that any two vertices are connected by a unique path and there is one special vertex r called root. The *parent* of a vertex v in T is the vertex connected to it on the path to the root. A *child* of v is a vertex of which v is the parent. The set of all children of v is denoted by $N^+(v)$. The subtree of T rooted at v is denoted by $T(r)$. The *depth* of a vertex is the length of its unique path to the root. The *height* of a vertex is the length of the longest downward path to a leaf from that node.

In order to avoid confusions, we use the term *node* exclusively when referring to trees. For a given recursive algorithm, in its recursion tree T, each node $x \in T$ corresponds to a call of the algorithm, each $y \in N^+(x)$, child of x, corresponds to a recursive call done inside (the call corresponding to) x, and the root is the initial call to the algorithm. We will use the terms node (of the recursion tree), call (to the algorithm) and iteration (of the algorithm) interchangeably. Moreover, when analysing the time complexity of recursive algorithms, we consider that the cost of an iteration does not include the cost of its recursive calls.

1.7 Structure of the Work

The work is structured as follows: in Chap. 2, we overview the main issues related to enumeration problems and the main techniques to design algorithms and proving their complexity; in Chap. 3, we overview the main kinds of biological networks and we highlight the dynamical structure of the biological networks: we argue the importance of enumeration algorithms for biological network analysis; in the subsequent chapters we show some examples of enumeration algorithms related to biological problems: in Chap. 4 we discuss the problem of enumerating stories, in Chap. 5 we discuss the problem of enumerating bubbles, and in Chap. 6 we discuss the problem of enumerating cycles or paths. Additionally, in Chap. 7 we discuss the problem of enumerating central and peripheral vertices. We conclude in Chap. 8, summarizing and reporting some open problems.

Part I
Enumeration Algorithm Techniques and Applications

Chapter 2
Enumeration Algorithms

2.1 Introduction

The aim of enumeration is listing all the feasible solutions of a given problem. For instance, given a graph $G = (V, E)$, enumerating all the paths or the shortest paths from a vertex $s \in V$ to a vertex $t \in V$, enumerating cycles, or enumerating all the feasible solutions of a knapsack problem, are classical examples of *enumeration problems*. An enumeration algorithm solves an enumeration problem.

While an optimization problem aims to find just the best solution according to an objective function, i.e. an extreme case, an enumeration problem aims to find all the solutions satisfying some constraints, i.e. local extreme cases. This is particularly useful whenever the objective function is not clear: in these cases, the best solution should be chosen among the results of the enumeration.

Moreover, sometimes it can be interesting to capture local structures of the data, instead of the global one, so that enumerating all remarkable local structures becomes particularly helpful.

In such a context, a good model is the result of a tradeoff between the size and the number of the solutions: whenever the sizes of the solutions are huge, it is more desirable to have relatively few solutions. For these reasons, the models usually include some parameters (such as solution size, frequency, and weight) or unify similar solutions.

It is worth observing that the number of solutions increases with the size of the input. Whenever this size is small, brute force algorithms are helpful, and simple implementations can successfully solve the problem. On the other hand, for large-scale data more sophisticated approaches from algorithm theory are required in order to guarantee a bounded increase of computation time when the input size increases.

In this chapter, we will present an overview of the main computational issues related to enumeration problems and the main techniques to design algorithms and to prove their complexity. These are part of the lecture notes, written together with Gustavo A.T. Sacomoto, during the lectures given by Takeaki Uno at the school on Enumeration Algorithms and Exact Methods (ENUMEX) in Bertinoro, Italy, on September 25–26th, 2012.

A. Marino, *Analysis and Enumeration*, Atlantis Studies in Computing 6,
DOI 10.2991/978-94-6239-097-3_2

Algorithm 1: BRUTEFORCE(i, X)

Input: An integer $i \geq 1$, a sequence of values $X = \langle x_0, \ldots, x_{i-1} \rangle$, eventually empty
Output: All the feasible sequences of length n whose prefix is X

```
1 if no solution includes X then return;
2 if i > n then
3 |  if X is a solution then output X;
4 else
5 |  foreach feasible value e of x_i do
6 |  |  BRUTEFORCE(i + 1, ⟨X, e⟩)
7 |  end
8 end
```

Structure of the Chapter
The chapter is structured as follows: in Sect. 2.2 we exploit the main algorithmic issues related to enumeration and we show some brute force approaches to solve them. In Sect. 2.3 we report the main technical framework to design efficient enumeration algorithms and in Sect. 2.4 we show the main amortization schema. In Sect. 2.5, we briefly discuss the tractability of enumeration problems in practice.

2.2 Algorithmic Issues and Brute Force Approaches

The design of enumeration algorithms involves several aspects that need to be taken into account in order to achieve correctness and effectiveness. Indeed, any enumeration algorithm has to guarantee that each solution is output exactly once, i.e. should avoid duplication. A straightforward way to achieve this is to store in memory all solutions already found, and whenever a new solution is encountered, test whether it has been already output or not. Clearly, this approach can be memory inefficient when the solutions are large with respect to the memory size, or there are too many of them. Dealing with this would require dynamic memory allocation mechanism and efficient search (*hash*). For these reasons, deciding whether a solution has been already output without storing the solutions already generated is a more suitable strategy that many enumeration algorithms try to apply.

Besides that, there are cases in which implicit forms of duplication should also be avoided, i.e. avoid outputting isomorphic solutions. To this aim, it is often useful to define a canonical form of encoding for the solutions allowing easy comparisons. The canonical form should provide a one-to-one mapping between the objects and their representation, without increasing drastically their size. In this way the problem of enumerating certain objects is turned into the enumeration of their canonical forms. However, in some cases, like graphs, sequence data and matrices, checking isomorphism is hard even by defining a canonical form. Nonetheless, in these cases the isomorphism can be still checked by using exponential algorithms that in practice turn out to be often efficient when the number of solutions is small.

Algorithm 2: BRUTEFORCE(X, D)

Input: A pattern X, a reference to a global database D
Output: All the patterns containing X not isomorphic between them and to any pattern contained in D

```
1 D ← D ∪ {X}
2 if no solution includes X then return;
3 if X is a solution then output X;
4 foreach X′ obtained by adding an element to X do
5 |   if ∄Z ∈ D such that Z isomorphic to X′ then
6 |   |   BRUTEFORCE(X′, D)
7 |   end
8 end
```

Simple structures, such as cliques and paths are generally easy to enumerate, since cliques can be obtained by iteratively adding vertices, and the set of paths can be easily partitioned. More complex structures, such as maximal (nothing can be added to the solution without losing some required property) or minimal (nothing can be subtracted from the solution without losing some required property) structures, or constrained structures, are more difficult to enumerate. In these cases, even if a solution can be found in polynomial time, the main issue is designing a way to generate other solutions from a given one, i.e. *defining a solution neighbourhood*, in order to allow visiting all the solutions by moving iteratively through the neighbourhoods.

It should be noted that using an exponential time approach to find each neighbour or having an exponential number of neighbouring solutions, can lead to time inefficiency. When an exponential number of possible choices have to be applied to a solution in order to possibly obtain other solutions, the enumeration process can take an exponential time for each solution, since there is no guarantee that any choice leads to a solution. For example this is very often the case concerning maximal solutions: removing some elements and adding others to get maximality allows to move iteratively to any solution, but, when the number of these combinations is exponential, the final cost per solution is also exponential. In such a context, if possible, restricting the number of neighbours of a solution or applying some pruning strategy to avoid redundant computation, can lead to more efficiency.

More complex cases concern the problems in which even finding a solution is NP-complete, such as SAT or Hamiltonian cycle. Nonetheless, in these cases, heuristics often effectively apply, specially when the problem turn out to be *usually easy*, like SAT, the *solutions are not huge*, like maximal and minimal structure enumeration, and the *size of the solution space is bounded*.

When the instance sizes are small, another approach to these problems, is to use brute force algorithms. For example, using a divide and conquer approach to enumerate all the candidates and selecting all feasible solutions, or by enlarging the solutions one by one and removing the isomorphic ones. Two basic schemas for brute force algorithms are informally described in Algorithms 1 and 2. In Algorithm 1 every solution is seen as an ordered sequence of values: by invoking BRUTEFORCE(1,$\emptyset$),

the feasible values are recursively found by enlarging the current solution; in this case, just the test whether X is a solution or not is required. Also Algorithm 2 tries to enlarge the current solution, but at each step we check whether the current solution has been already considered in the past computation: the result of the past computation is stored in a database D.

Note that for both the algorithms, it is necessary to know how to transform a candidate X into another candidate X'. Moreover, it is worth observing that, in both cases, an accurate a priori checking whether X is contained in any solution or not could save a lot of useless computation.

2.3 Basic Algorithms

Since the number of solutions of many enumeration problems are usually exponential in the size of the instance, enumeration algorithms require often at least exponential time. On the other hand, it is quite natural to ask for a polynomial time algorithm whenever the number of solutions is polynomial. In such a context, the complexity classes of enumeration problems are defined depending on the number of solutions, so that if the number of solution is small, an efficient algorithm has to terminate after short (polynomial) time, otherwise it is allowed to spend more time. According to this idea, the following complexity classes have been defined [1].

Definition 2.1 An enumeration algorithm is *polynomial total time* if the time required to output all the solutions is bounded by a polynomial in the size of the input and the number of solutions.

Definition 2.2 An enumeration algorithm is *polynomial delay* if it generates the solutions, one after the other in some order, in such a way that the delay until the first is output, and thereafter the delay between any two consecutive solutions, is bounded by a polynomial in the input size.

Intuitively, the polynomial total time definition means that the delay between any two consecutive solutions has to be polynomial on the average, while the polynomial delay definition implies that the maximum delay has to be polynomial. Hence, Definition 2.2 implies Definition 2.1.

For a comprehensive catalogue of known enumeration algorithms and their classification we invite the reader to see [24].

The basic technique for designing enumeration algorithms are: backtracking (depth-first search with lexicographic ordering), binary partition (branch and bound like recursive partition algorithm), reverse search (search on traversal tree defined by parent-child relation). The rest of this section is devoted to exploit the features of these schemas. It is worth observing that this categorization is not strict, since very often these technique overlap each other.

2.3.1 Backtracking

A set $F \subseteq 2^U$ (of subsets of U) satisfies the *downward closure* if for any $X \in F$ and for any $X' \subseteq X$, we have $X' \in F$, in other words, for any X belonging to F we have that any subset of X also belongs to F. Given a set U and an oracle to decide whether $X \subset U$ belongs to F, an unknown set of 2^U satisfying the downward closure, we consider the problem of enumerating all (maximal) elements of F. The backtracking technique is mainly applied to these problems. In this approach by starting from an empty set, the elements are recursively added to a solution. The elements are usually indexed, so that in each iteration, in order to avoid duplication, only an element whose index is greater than the current maximum element is added. After all the examinations concerning one element, by backtracking, all the other possibilities are exploited. The basic schema of backtracking algorithms is shown by Algorithm 3. Note that whenever it is possible to apply this schema, we obtain a polynomial delay algorithm, whose space complexity is also polynomial. The technique proposed relies on a depth-first search approach. However, it is worth observing that in some cases of enumeration of families of subsets exhibiting the downward closure property, arising in the mining of frequent patterns (e.g., mining of frequent itemsets), besides the depth-first backtracking, a breadth-first approach can be also successfully used. For instance this is the case of the Apriori algorithm for discovering frequent itemsets [25].

Algorithm 3: BACKTRACK(S)

Input: $S \subseteq U$ a set (eventually empty)
Output: All the solutions containing S

```
1 output S
2 Let π(x) be the index associated to an element x ∈ U
3 foreach e > max_{x∈S} π(x) do
4 |   if S ∪ {e} is a solution then
5 |   |   BACKTRACK(S ∪ {e})
6 |   end
7 end
```

Algorithm 4: SUBSETSUM(S)

Input: S a set (eventually empty) of integers belonging to the collection $U = \{a_1, \ldots, a_n\}$
Output: All the subsets of U containing S whose sum is less than b.

```
1 ouput S
2 Let π(x) be the index associated to an element x
3 foreach i > max_{x∈S} π(x) do
4 |   if a_i + Σ_{x∈S} x < b then
5 |   |   SUBSETSUM(S ∪ {a_i})
6 |   end
7 end
```

2.3.1.1 Enumerating All the Subsets of a Collection $U = \{a_1, \ldots, a_n\}$ Whose Sum is Less Than b

By using the backtracking schema, it is possible to solve the problem as shown by Algorithm 4. Each iteration outputs a solution, and take $O(n)$ time, so that we have $O(n)$ time per solution. It is worth observing that if we sort the elements of U, then each recursive call can generate a solution in $O(1)$ time, so that we have $O(1)$ time per solution.

2.3.2 Binary Partition

Let X be a subset of F, the set of solutions, such that all elements of X satisfy a property P. The binary partition method outputs X only if the set is a singleton, otherwise, it partitions X into two sets X_1 and X_2, whose solutions are characterized by the disjoint properties P_1 and P_2 respectively. This procedure is repeated until the current set of solutions is a singleton. The bipartition schema can be successfully applied to the problem of enumeration of paths of a graph connecting two vertices s and t, of the perfect matchings of a bipartite graph [26], of the spanning trees of a graph [27]. If every partition is non-empty, i.e. all the internal nodes of the recursion tree are binary, we have that the number of internal nodes is bounded by the number of leaves. In addition, if we have that the partition oracle takes polynomial time, since every leaf outputs a solution, we have that the resulting algorithm is polynomial total time. On the other hand, even if there are empty partitions, i.e. internal unary nodes in the recursion tree, if the height of tree is bounded by a polynomial in the size of the input and the partition oracle takes polynomial time, then the resulting algorithm is polynomial delay.

2.3.2.1 Enumerating All the (s, t)-Paths in a Graph $G = (V, E)$

The partition schema chooses an arc $e = (s, r)$ incident to s, and partitions the set of all the (s, t)-paths into the ones including e and the ones not including e. The (s, t)-paths including e are obtained by removing all the arcs incident to s, and enumerating the (r, t)-paths in this new graph, denoted by $G - s$. The (s, t)-paths not including e are obtained by removing e and enumerating the (s, t)-paths in the new graph, denoted by $G - e$. The corresponding pseudocode is shown by Algorithm 5. It is worth observing that if the arc e is badly chosen, a subproblem could not generate any solution; in particular, the set of the (r, t)-paths in the graph $G - s$ is empty if t is not reachable from r, while the set of the (s, t)-paths in $G - e$ is empty if t is not reachable from s. Thus before performing the recursive call to the subproblems it could be useful to test the validity of e, by testing the reachability of t in these modified graphs. Notice that the height of the recursion tree is bounded by $O(|V| + |E|)$, since at every level the size of the graph is reduced by one vertex

or arc. The cost per iteration is $O(|V| + |E|)$, the reachability test. Therefore, the algorithm has $O((|V| + |E|)^2)$ delay or $O(|E|^2)$ delay for connected graphs.

This problem has been studied in [9, 28, 29], and in [10], guaranteeing a linear delay. In Chap. 5 we will modify this latter algorithm in order to enumerate *bubbles*. In the particular case of undirected graphs, in Chap. 6 we will show an algorithm based on this bipartition approach having an output sensitive amortized complexity, as shown in [16]. In the particular case of shortest paths, the enumeration problem has been studied in [30]. It is worth observing that the problem of enumerating all the (s, t)-paths in a graph is equivalent to the problem of enumerating all the cycles passing through a vertex.

Algorithm 5: PATHS(G, s, t, S)

Input: A graph G, the vertices s and t, a sequence of vertices S (eventually empty)
Output: All the paths from s to t in G

```
1  if s = t then
2  |  output S
3  |  return
4  end
5  choose an arc e = (s, r)
6  if no (r, t)-path in G − s then
7  |  PATHS(G − e, s, t, S)
8  |  return
9  end
10 if no (s, t)-path in G − e then
11 |  PATHS(G − e, r, t, ⟨S, s⟩)
12 |  return
13 end
14 PATHS(G − s, r, t, S)
15 PATHS(G − e, s, t, S)
```

2.3.3 Reverse Search

The reverse search schema defines for any solution a solution called *parent solution* [31], in a way that this parent-children relationship does not induce a cyclic graph or DAG, but induces a tree. In this way, in order to enumerate all the solutions, it is sufficient to traverse the tree by performing a depth first search, so that the number of iterations is equal to the number of solutions. It is worth observing that the tree induced by the parent child relationship does not need to be stored in memory, but it is sufficient to use an algorithm for finding all the children of a parent. Moreover it could be preferable to have an algorithm able to find the $(i + 1)$th child of a node, given the ith child.

Since the number of iterations is equal to the number of solutions, we have that the cost per solution is equal to the cost per iteration. Thus if finding the next child of

a node costs $O(f(n))$ time, where n is the input size, the resulting computation time per iteration is $O(f(n))$. Hence the algorithm is polynomial total time whenever $f(n)$ is polynomial. The space complexity is given by the memory usage of an iteration and by the height of the depth first search tree. This latter cost is not required when we have an algorithm able to find the $(i+1)$th child of a node, given its ith child. The delay between two successive solutions is also $O(f(n))$ by using the alternative output technique [32].

Indeed alternative output technique aims to reduce the delay, by avoiding that the depth first search backtrack along long paths without outputting any solution. As shown by Algorithm 7 the solutions are outputted before the recursive calls when the current depth first search level is even, otherwise, i.e. in the odd levels, the solutions are output after the recursive calls. In this way for any two successive solutions we have a delay at most $2f(n)$, where $f(n)$ is the cost of an iteration. Indeed suppose that the parent child relationship induces a path of solutions $x_1, \ldots, x_{g(n)}$ and there is a solution $x_{g(n)+1}$ that is a child of x_1, where $g(n)$ is a function of n. If the cost per iteration is $O(f(n))$, by applying Algorithm 6, for any i with $1 \leq i \leq g(n)$, the delay is $O(f(n))$, and the delay between x_k and x_{k+1} is $O(g(n))$. By applying Algorithm 7, by supposing $g(n)$ odd, the solutions are generated in the following order $x_2, x_4, \ldots x_{g(n)-1}, x_{g(n)}, x_{g(n)-2}, x_{g(n)-4}, \ldots, x_3, x_1, x_{g(n)+1}$, so that the delay is $O(2 \cdot f(n)) = O(f(n))$.

In conclusion, by applying this technique, every time an enumeration algorithm takes $O(f(n))$ time in each iteration and also outputs a solution on each iteration, the delay $O(f(n))$ can be turned into a worst case delay $O(f(n))$.

Algorithm 6: REVERSESEARCH(S)

```
1 output S
2 foreach child S' of S do
3 |   REVERSESEARCH(S')
4 end
```

Algorithm 7: ALTERNATIVEOUTPUT(S, *depth*)

Input: A solution S, an integer $depth$
Output: All the solutions descendants of S in the tree induced by the parent-child relationship

```
1 if depth is even then output S;
2 foreach child S' of S do
3 |   ALTERNATIVEOUTPUT(S, depth + 1)
4 end
5 if depth is odd then output S;
```

2.3.3.1 Maximal Clique Enumeration

A clique is a complete graph, i.e. a graph in which any two vertices are connected. Finding the clique of maximum size in a graph $G = (V, E)$ is NP-hard [33], while finding a maximal clique is an easy task that can be solved in $O(|E|)$ time: by starting with an arbitrary clique (for instance, a single vertex), grow the current clique one vertex at a time, adding it if connected to each vertex in the current clique, and discarding it otherwise. The clique enumeration problem is the problem of enumerating all the complete subgraph of a given graph in input. This problem has been widely studied by [34–36]. The bipartite clique enumeration problem is the problem of enumerating all the complete bipartite subgraphs of a bipartite graph and it can be efficiently reduced to a clique enumeration problem [34].

It is worth observing that the set of cliques is *monotone*, since any subset of the vertices of a clique is also a clique. This means that the backtracking technique can be successfully applied. Checking whether a recursive call is going to produce at least a clique costs $O(|E|)$ time, and has to be repeated for at most $|V|$ recursive calls, so that the final cost is $O(|V||E|)$ per clique.

When the number of solutions increase exponentially when the size of the instance input increases linearly, it seems hard post-processing the solutions found, so that often the simple enumeration problem is turned in enumeration of maximal structures. In this way, the solution set becomes not redundant. More formally, a solution X is maximal if for any $X \subset X'$, X' is not a solution. In general the problem of finding maximal solutions is more difficult, since it is often harder to find a *neighbourhood* relationship between them. However there are some exceptions, like enumerating maximal clique.

Also in real contexts it seems more promising enumerating all the maximal cliques instead of all the cliques: it has been estimated that in real world graphs, even if they are sparse and the size of their cliques is small, the number of maximal cliques is between 0.1 and 0.001 % the number of its cliques (see also [37]). Moreover, restricting the enumeration to maximal cliques does not lead to lose any information since any clique is included in at least one maximal clique.

Given a graph $G = (V, E)$, whose vertices are indexed, a set of vertices $X \subseteq V$ is said to be *lexicographically* greater than $Y \subseteq V$ if the vertex whose index is minimum in $(X \setminus Y) \cup (Y \setminus X)$ is contained in X. Moreover, for any $X, Y \subseteq V$, the trichotomy property holds, i.e. exactly one of the following holds: $X < Y$, $X = Y$, or $Y > X$. For any vertex set S, we define $S_{\leq i}$ as $S \cap \{v_1, \ldots, v_i\}$.

Let $C(K)$ be the lexicographically smallest maximal clique including a clique $K \subseteq V$, $C(K)$ can be computed by greedily adding vertices to K in lexicographic order of the indices. Observe that for any set K, $C(K)$ is not lexicographically smaller than K.

Given a maximal clique K we define the parent of K, $P(K)$, as $C(K_{\leq i-1})$, such that i is the maximum index satisfying $C(K_{\leq i-1}) \neq K$. Notice that $C(K_{\leq i-1})$ can be efficiently computed by removing the vertices from K by starting from the ones whose index is greater and computing C on the remaining vertices while $C(K) = K$ holds. The lexicographically smallest clique, denoted as K_0, has no parent. Since for

Algorithm 8: EnumMaximalCliques(G, K)

Input: A graph $G = (V, E)$, a maximal clique $K \subseteq V$
Output: All the maximal cliques descendants of K in the tree induced by the parent-child relationship between maximal cliques

```
1 output K
2 foreach vertex v ∈ V not in K do
3 |   K' ← K[v]
4 |   if P(K') = K then
5 |   |   EnumMaximalCliques(G, K)
6 |   end
7 end
```

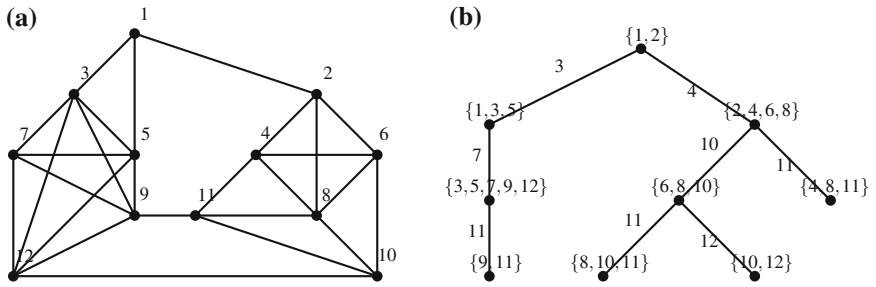


Fig. 2.1 A graph and the recursion tree induced by Algorithm 8

any K, $P(K)$ is lexicographically greater than K, and $P(K)$ is uniquely defined, the parent-child relationship induces an acyclic graph, that is a tree (Fig. 2.1).

For any maximal clique K and any vertex v_i, we define $K[v_i]$ as $C((K_{\leq i} \cap N(v_i)) \cup \{v_i\})$, where $N(v_i)$ is the neighbourhood of v_i. Thus a maximal clique K' is a child of the maximal clique K, if there exists v_i, with $v_i \notin K$, such that $K' = K[v_i]$. Hence in order to compute the children of a maximal clique K, it is sufficient to check for any v_i whether $P(K[v_i])$ is equal to K.

Observe that for any maximal clique K, $C(K)$ and $P(K)$ can be computed in $O(|E|)$ time. All children of K can be found by at most $|V|$ tests, so that the cost of each iteration is bounded by $O(|V||E|)$ time. Thus, since the number of iterations is equal to the number of solutions, the final cost is $O(|V||E|)$ per maximal clique.

2.3.3.2 Non-Isomorphic Ordered Tree Enumeration

Several enumeration problems aim to enumerate all the *substructures* of a given instance, like paths of a graph. However, applications sometimes require solutions satisfying certain constrains, like enumerating path or cycles of a fixed length or enumerating the cliques of a given size. Other problems instead aim to find all the

structures of a given class, like enumerating the permutations of size n, enumerating trees, crossing lines in a plane, matroids, and binary matrices. Enumerating non trivial structures often implies enumerating non isomorphic structures. In general two structures are isomorphic whenever it is defined a one-to-one correspondence between their elements. For instance a circular sequence is isomorphic to another if and only if it can be transformed in it by using a rotation, a matrix is isomorphic to another matrix if and only if each one can be transformed in the other one by swapping rows and columns, a graph is isomorphic to another graph if and only if their adjacency matrices are isomorphic, i.e. there is a one to one mapping between their vertices that preserves the adjacency.

Let us consider the problem of enumerating ordered trees, trees in which the ordering of the children of each vertex is specified. The isomorphism between two ordered trees is inductively defined as follows: two leaves are isomorphic; two trees rooted on x and y, whose order lists of children are $\langle x_1, \ldots, x_p \rangle$ and $\langle y_1, \ldots, y_q \rangle$ respectively, are isomorphic if $p = q$ and for any i, with $1 \leq i \leq p = q$, the subtree rooted on x_i is isomorphic to the subtree rooted on y_i. This problem has been studied in [38], and by fixing the number of leaves in [39].

Given an ordered tree, we define the indexing of its vertices as the visiting order of a left-first DFS, i.e. a depth first search that visits the children of a vertex following their order. This indexing procedure is unique and isomorphism between two ordered trees, whose vertices are indexed as described, can be checked comparing the edge sets: the two indexed trees are isomorphic if and only if they have the same edge set.

Moreover, the left-first DFS can be used to encode the ordered trees. To this aim, we define the depth sequence as $\langle h_1, \ldots, h_n \rangle$, where h_i is the depth of vertex v_i in the left-first DFS tree, where v_i is the ith vertex visited by a left-first DFS. There is a one-to-one correspondence between the ordered trees and the depth sequences, so that isomorphism can be checked by comparing the depth sequences, as shown by Fig. 2.2.

By following the reverse search schema, we define the parent-child relationship between non-isomorphic trees. In particular the parent of an ordered tree is defined by the tree, obtained by removing the vertex having the largest index, i.e. by removing from a depth sequence its last element (the last element visited by a left-first DFS). Recall that the indexing induced by the left-first DFS is such that the largest index is the leaf of the rightmost branch of the tree. Observe that the size of the parent is

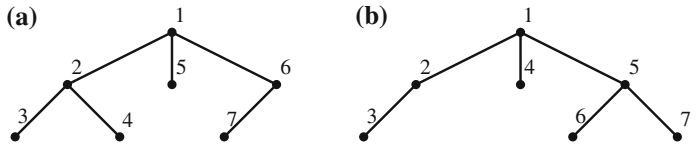


Fig. 2.2 Two non isomorphic ordered trees, labelled by using left-first DFS, and their depth sequences. **a** $\langle 0, 1, 2, 2, 1, 1, 2 \rangle$. **b** $\langle 0, 1, 2, 1, 1, 2, 2 \rangle$

smaller than the size of the children, any ordered tree have exact one parent, except the empty tree, so that the relationship induces an acyclic graph.

For any ordered tree T, whose depth-sequence is $\langle h_1, \ldots, h_n \rangle$, the children of T according to the parent-child relationship defined before, are all the ordered trees obtained by adding a new vertex v_{n+1} as the rightmost child of a vertex belonging to the rightmost path. Let h_{n+1} be the depth of the new vertex v_{n+1}. Since h_n is the rightmost leaf of T, we have that it belongs to the rightmost path, to be precise, v_n is the last vertex of this path. Thus, the depths of the vertices of the rightmost path of T, from the root to v_n, are exactly the interval $[0, h_n]$. Since the new vertex v_{n+1} is a child of a vertex in this path, the depth h_{n+1} is in the interval $[1, h_n + 1]$. Thus the children of an ordered tree T, with depth-sequence $\langle h_1, \ldots, h_n \rangle$, are all the ordered trees whose depth sequence is $\langle h_1, \ldots, h_n, h_{n+1} \rangle$, with $1 \leq h_{n+1} \leq h_n + 1$. An example is given in Fig. 2.3.

By using these observations, we can enumerate all the ordered trees of size less than k, as shown by Algorithm 9. Notice that the inner loop takes constant time, so that the time complexity is $O(1)$ per solution.

Algorithm 9: EnumOrderedTree(T, k)

Input: A tree T (eventually empty) and an integer k
Output: All the non-isomorphic trees of size at most k, whose depth sequence contains as prefix the depth sequence of T

```
1 output T
2 if size of T = k then return;
3 foreach vertex v in the right most path do
4 |   Let T' be the tree obtained from T by adding a rightmost child to v
5 |   EnumOrderedTree(T', k)
6 end
```

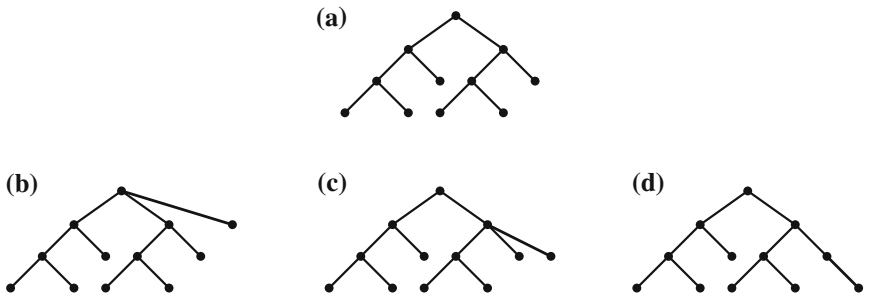


Fig. 2.3 An ordered tree, its depth sequence (**a**), and its children with their depth sequences (**b**), (**c**) and (**d**). **a** ⟨0, 1, 2, 3, 3, 2, 1, 2, 3, **2**⟩. **b** ⟨0, 1, 2, 3, 3, 2, 1, 2, 3, **2**, *1*⟩. **c** ⟨0, 1, 2, 3, 3, 2, 1, 2, 3, **2**, *2*⟩. **d** ⟨0, 1, 2, 3, 3, 2, 1, 2, 3, **2**, *3*⟩

2.3.3.3 Non-Isomorphic Tree Enumeration

We now consider the problem of enumerating non-ordered trees, i.e. trees in which the ordering of the children of each vertex is not specified. The isomorphism between two (non-ordered) trees is inductively defined as follows: two leaves are isomorphic; two trees rooted on x and y, whose children lists are X and Y respectively, are isomorphic if $|X| = |Y| = p$ and there exist two permutations of X and Y, $\langle x_1, \ldots, x_p \rangle$ and $\langle y_1, \ldots, y_p \rangle$ respectively, such that for any i, with $1 \leq i \leq p$, the subtree rooted on x_i is isomorphic to the subtree rooted on y_i. This problem has been studied in [40], by fixing the diameter in [41], and in the more general case of coloured rooted trees in [42].

The näive approach, to use the same algorithm for ordered tree enumeration to enumerate non-ordered trees, would produce many duplicate solutions, since each non-ordered tree may correspond to an exponential number of ordered trees. Which in turn, would be very inefficient.

In order to define the canonical form of representation of a rooted tree, we use its *left-heavy* embedding, defined as the lexicographically maximum depth sequence among all the ordered trees corresponding to T (Fig. 2.4). Therefore, two non-ordered rooted trees are isomorphic if and only if they have the same left-heavy embedding.

The parent child relationship between canonical forms is defined as follows: the parent of a left-heavy embedding is obtained by the removal of the rightmost leaf of the corresponding tree, the same for ordered trees. Observe that the parent t' of a left-heavy embedding t of T is a left-heavy embedding too, otherwise there would be another sequence greater than t' such that by adding back the rightmost leaf of T we would obtain a depth sequence for T that is lexicographically greater than t (Fig. 2.5).

Hence any child of a rooted tree T is obtained by adding a vertex as children of the vertices belonging of the rightmost path, like for ordered trees. However, some trees obtained by adding a vertex in this way are not children of T, since the resulting sequence does not coincide with their left-heavy embedding. This can happen if there exists a vertex x in the rightmost path of T, such that the depth sequence $t = \langle s_1, \ldots, s_p \rangle$ of $T(r)$, where r is the rightmost child of x, is a prefix of the depth sequence $t' = \langle s_1, \ldots, s_p, \ldots s_q \rangle$ of $T(r')$, where r' is the second rightmost child of x, so that the depth sequence of T ends with t concatenated with t'. Indeed, in this case, by adding a vertex at depth y to $T(r)$ and obtaining $t'' = \langle s_1, \ldots, s_p, y \rangle$

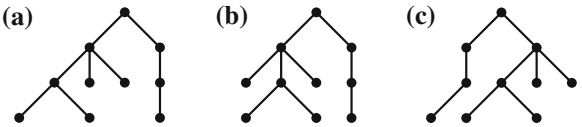


Fig. 2.4 Three isomorphic rooted tree and their depth sequences. The first one is the left heavy embedding. **a** $\langle 0, 1, 2, 3, 3, 2, 2, 1, 2, 3 \rangle$. **b** $\langle 0, 1, 2, 2, 3, 3, 2, 1, 2, 3 \rangle$. **c** $\langle 0, 1, 2, 3, 1, 2, 3, 3, 2, 2 \rangle$

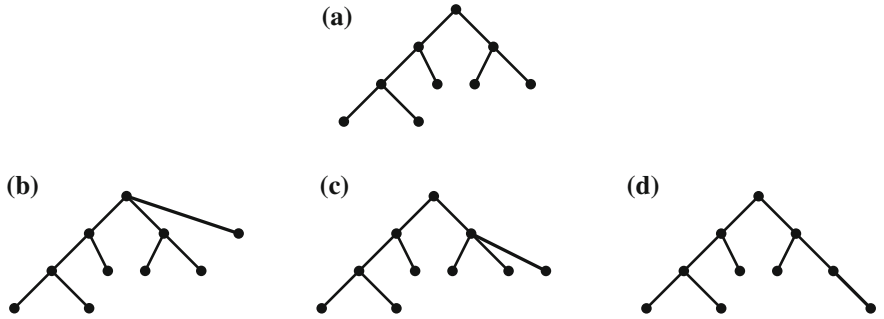


Fig. 2.5 An rooted tree, and its depth sequence (**a**). **b** and **c** are its children, while **d** is not a child of (**a**). **a** ⟨0, 1, 2, 3, 3, 2, 1, 2, 3, 2⟩. **b** ⟨0, 1, 2, 3, 3, 2, 1, 2, 2, *1*⟩. **c** ⟨0, 1, 2, 3, 3, 2, 1, 2, 2, *2*⟩. **d** ⟨0, 1, 2, 3, 3, 2, 1, 2, 2, *3*⟩

as depth sequence of $T(r)$, the depth sequence of T ends with t concatenated with t'': if s_{p+1} is lexicographically smaller than y, this is not a leaf-heavy embedding, since the depth sequence ending with t'' concatenated with t is lexicographically greater. Thus, since s_{p+1} is the depth of the rightmost leaf of $T(r')$, to get all and just the children of T, we have to consider all the possible ways to add a vertex as children of a vertex belonging to the rightmost path, so that its depth is smaller or equal to s_{p+1}.

The *copy vertex* is thus defined as the highest (lowest depth) vertex x in T with at least two[1] children, r and r' (the rightmost and the second rightmost child respectively), such that the depth sequence $\langle s_1, \ldots, s_p \rangle$ of $T(r)$ is a prefix of the depth sequence $\langle s_1, \ldots, s_p, \ldots s_q \rangle$ of $T(r')$. Given a tree T with copy vertex x, in order to generate the children of T, we have to consider two cases: the prefix of the depth sequences is proper or the depth sequences are equal. In the first case, there exists s_{p+1} and by attaching a new rightmost child to a vertex v, with depth $\leq s_{p+1}$, in the rightmost path of T we obtain a new tree T' that is also a left-heavy embedding. Moreover, the new copy vertex of T' is v, if the depth v is not equal to the depth of x; or x, otherwise. On the other case, the subtrees $T(r)$ and $T(r')$ are equal and by attaching a new rightmost child to a vertex v, with depth smaller or equal to the depth of x, in the rightmost path of T we obtain a new tree T' that is also a left-heavy embedding, and the new copy vertex of T' is v. In both cases, we are able to generate the new tree T' and update the copy vertex in constant time. The algorithm is shown by Algorithm 10. Each iteration of the loop costs $O(1)$, so that we have a final cost of $O(1)$ per solution.

[1] If T is a path, the copy vertex is defined as the root.

Algorithm 10: EnumRootedTree(T, x)

Input: A tree T (eventually empty), an integer k, and a vertex x
Output: All the non-isomorphic rooted trees of size at most k, whose depth sequence contains as prefix the depth sequence of T

```
1  output T
2  if size of T = k then return;
3  r ← the rightmost child of x
4  r' ← the second rightmost child of x
5  if depth sequence of (T(r')) ≠ depth sequence of T(r) then
6  |  y ← the vertex of T(r') after the prefix T(r)
7  else
8  |  y ← x
9  end
10 foreach vertex v in the rightmost path of T, in increasing depth order do
11 |  add a rightmost child to v
12 |  if depth of v = depth of y then
13 |  |  EnumRootedTree(T, x)
14 |  |  break
15 |  end
16 |  EnumRootedTree(T, v)
17 |  remove the rightmost child of v
18 end
```

2.4 Amortized Analysis

In this section, we explore techniques to analyse the running time of a certain kind of enumeration algorithms. Specifically, enumeration algorithms with a tree-shaped recursion structure.

Suppose a enumeration algorithm with a tree-shaped recursion structure takes $O(n)$ time per node. Based only on this, it is not possible to polynomially bound the time spent to output each solution. We can have exponentially many nodes and a small number of solutions as in, for example, the enumeration of feasible solutions of SAT using a branch-and-bound algorithm. However, if every node outputs a solution, then algorithm takes $O(n)$ per solution. Now, suppose that each leaf outputs a solution and each node takes $O(n)$ time. Again, this is not enough to polynomially bound time per solution, since we can have an exponential number of internal nodes and only few leaves. In addition, we need that either the height of the tree is bounded, in this case the number of nodes is bounded by the number of solutions (leaves) times the height; or each internal node has at least two children, the number of nodes is bounded by two times the number of solutions.

These three scenarios: every node outputs a solution, every leaf outputs a solution and the height of the tree is bounded, and every leaf outputs a solution and each internal nodes has at least two children, are the typical ones in which we can polynomially bound the time complexity. In each case, the time complexity per solution depends on the maximum time complexity $O(n)$ over all nodes. In order to do

better, we have to use amortized analysis. The rest of the section is devoted to three amortized analysis techniques: basic amortization, amortization by children and push out amortization.

2.4.1 Basic Amortization

A recursive enumeration algorithm usually solves the problem by breaking it into subproblems, which are generally smaller, in both input and output size, than the original problem. The recursion tree for this case has many bottom level nodes taking a short time and a fewer nodes closer to the root taking a long time. We call this effect in the recursion tree *bottom-wideness*. However, this observation alone is not enough to provide good amortized bounds. For instance, Fig. 2.6a, b have bottom level nodes (leaves) taking $O(1)$, but the amortized complexity is still $O(n)$, the maximum cost among the nodes. In both cases, there were a sudden decrease in the computation times.

In the tree of Fig. 2.6c each internal node has two children, all the nodes in the same level have the same cost, and the cost of each node decreases by a constant at each level. It is not hard to show that the average complexity per node in this case is $O(1)$. That is, there was a reduction from $O(n)$ to $O(1)$ when considering the amortized complexity. Lemma 2.2 presents a generalization of this example, every node has two children and the costs are proportional to the height of the node. Technical Lemma 2.1 is used in the proof of the Lemma 2.2.

Lemma 2.1 *For any polynomial* $p(x) = \sum_{k=0}^{m} a_k x^k$, *there exists* δ *and* $\alpha < 1$, *such that* $\frac{p(x+1)}{2p(x)} < \alpha$, *for all* $x > \delta$.

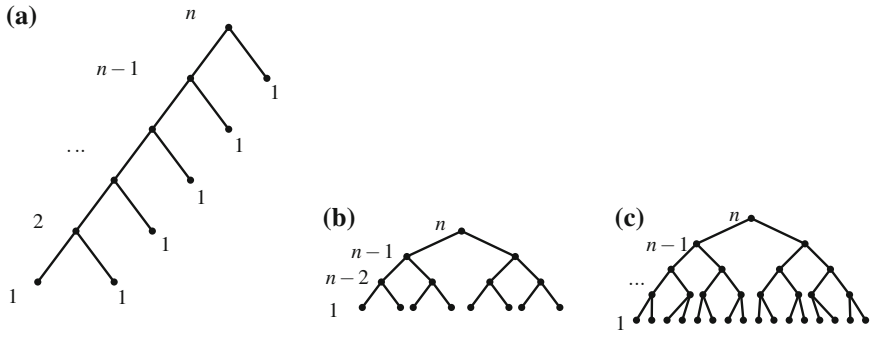


Fig. 2.6 Recursion trees with the cost of each node. **a** The cost of the internal nodes decrease by 1 and the all leaves have cost 1. **b** The nodes in the same level have the same cost, decreasing by 1 from n until $n - 2$. The leaves have cost 1. **c** The nodes in the same level have the same cost, decreasing by 1 from n until 1

Proof It is easy to prove that $\lim_{x\to\infty} \frac{p(x+1)}{2p(x)} = 1/2$. Thus, from the definition of limit, there exist ϵ and δ (depending only on ϵ), such $\frac{p(x+1)}{2p(x)} - 1/2 < \epsilon$ for all $x > \delta$. Choosing $\epsilon < 1/2$, we have that $\frac{p(x+1)}{2p(x)} < \epsilon + 1/2 = \alpha < 1$ for all $x > \delta$.

Lemma 2.2 *Let T be a recursion tree with height n, such that every internal node has degree 2; and the cost for each node is $O(p(i))$, where $p(i)$ is a polynomial and i is the height of the node. Then, the amortized cost for each node is $O(1)$.*

Proof The number of nodes with height i is 2^{n-i}, since the internal nodes have degree 2. The cost of each level i of the tree is bounded by $2^{n-i}\,p(i)$ and the total cost of the tree is $\sum_{i=1}^{n} 2^{n-i}\,p(i)$. Let us consider the ratio of the cost of two adjacent levels in the tree,

$$r(i) = \frac{2^{n-(i+1)}\,p(i+1)}{2^{n-i}\,p(i)} = \frac{p(i+1)}{2p(i)}.$$

By Lemma 2.1, $r(i) < \alpha < 1$, for all $i > \delta$. Implying that the cost of each level $i \geq \delta$ decrease by α and the sum of the costs of all levels $i \geq \delta$ is bounded by the sum of the geometric series, i.e.

$$\sum_{\delta \leq i \leq n} 2^{n-i}\,p(i) < 2^{n-\delta}\,p(\delta) \sum_{\delta \leq i \leq n} \alpha^{i-\delta} < 2^{n-\delta}\,p(\delta) \sum_{i=0}^{\infty} \alpha^{i} = \frac{2^{n-\delta}\,p(\delta)}{1-\alpha}.$$

Therefore, amortized cost of each node in the levels above δ is

$$\frac{\sum_{\delta \leq i \leq n} 2^{n-i}\,p(i)}{\sum_{\delta \leq i \leq n} 2^{n-i}} < \frac{2^{n-\delta}\,p(\delta)}{1-\alpha}\,\frac{1}{2^{n-\delta}} = \frac{p(\delta)}{1-\alpha} = O(1).$$

The last equality follows from the fact that α and $delta$ are constants. Moreover, the cost of the nodes with $i \leq \delta$ is also $O(1)$.

Theorem 2.1 is a straightforward generalization of Lemma 2.2.

Theorem 2.1 *Let T be a recursion tree with height n, such that every internal node has degree* at least *2; and the cost for each node is $O(p(i))$, where $p(i)$ is a polynomial and i is the height of the node. Then, the amortized cost for each node is $O(1)$.*

2.4.1.1 Enumerating Connectivity Elimination Orderings of a Connected Graph G

Given a connected graph $G = (V, E)$, a *connectivity elimination ordering* is an ordering of the vertices such that the removal of each vertex keeps the remaining graph connected. Algorithm 11 enumerates all connectivity elimination orderings.

Each call of the algorithm takes $O(|V|^3)$ time, since for each $v \in V$ it checks if $G - v$ is connected. Moreover, for any connected graph there are at least two vertices such that their removal maintain the graph connected. Therefore, the hypothesis of Theorem 2.1 are satisfied, and the amortized complexity per node is $O(1)$. Since, in this case, the number of nodes is at most 2 times the number of leaves, the amortized complexity per solution is also $O(1)$.

Algorithm 11: EnumOrderings($G = (V, E)$, X)

Input: A graph G and sequence X that is a prefix of a connectivity elimination order.
Output: The set of all connectivity elimination orders of G.

```
1 if V = ∅ then
2 |  output X
3 end
4 foreach v ∈ V do
5 |  if G − v is connected then
6 |  |  EnumOrderings(G − v, ⟨X, v⟩)
7 |  end
8 end
```

2.4.2 Amortization by Children

The basic amortization strategy presented in the previous section, in the form of Theorem 2.1, requires that every leaf has the same depth, and the cost of each node depends uniformly on the height. Though there are applications for Theorem 2.1, these requirements are global, they depend on the tree as whole, imposing a very strict structure in the recursion tree. In this section, we start developing amortization techniques with weaker hypothesis, so that they can be applied also in the case of more biased trees. For this, we focus on local tree structure. Theorem 2.2 presents a simple amortization scheme using only the parent-children structure.

Theorem 2.2 *Let T be a recursion tree and $T(x)$ the cost of node $x \in T$. The amortized cost for each node is $O(\max_{z \in T} \frac{T(z)}{|N^+(z)|+1})$.*

Proof We divide the cost $T(x)$ between the node x and its children $N^+(x)$. In this way the new cost of x is $\frac{T(x)}{|N^+(x)|+1}$ plus the cost received from its parent y. Thus,

$$\begin{aligned} T'(x) &= O\left(\frac{T(x)}{|N^+(x)|+1} + \frac{T(y)}{|N^+(y)|+1}\right) \\ &= O\left(\max_{z \in \{x,y\}} \frac{T(z)}{|N^+(z)|+1}\right) = O\left(\max_{z \in T} \frac{T(z)}{|N^+(z)|+1}\right). \end{aligned}$$

2.4.2.1 Enumerating All Simple Paths of G Starting from s

Given a graph $G = (V, E)$ and a vertex $s \in V$, we consider the problem of enumerating all simple paths of G that start on s. Algorithm 12 solves this problem. Each call outputs a solution and takes $O(|N(s)|)$ time, since we have to explore all edges from s. Moreover, each edge from s generates a recursive call. Therefore, applying Theorem 2.2 we have that the amortized cost per node is $O(\max_{z \in T} \frac{|N(z)|}{|N(z)|+1}) = O(1)$.

Algorithm 12: EnumPaths$(G = (V, E), s, \pi)$

Input: A graph G, a vertex s and a path π from s.
Output: The set of all paths from s in G with prefix π.

```
1 output π
2 foreach v ∈ N(s) do
3 |   EnumPaths(G − s, v, ⟨π, s⟩)
4 end
```

2.4.3 Push Out Amortization

In Lemma 2.2 the key property was that the total cost on each level increases with a constant factor, by going to the next deeper level. Intuitively, the increase of computation time is good because it forbids a sudden decrease, as the one of the trees in Fig. 2.6a, b. In this section, we apply the same idea locally. Instead of comparing the total cost of two adjacent levels we compare the cost of a node with the total cost of its children. Lemma 2.3 gives a precise statement for this local increase property. Afterwards, Theorem 2.3 generalizes Lemma 2.3, by combining it with the amortization by children of Theorem 2.2.

Lemma 2.3 *Let T be a recursion tree, such that the cost of each leaf is $O(T^*)$; and there exist $\alpha > 1$ such that every internal node $x \in T$ satisfy $\sum_{y \in N^+(x)} T(y) \geq \alpha T(x)$, where $T(x)$ is the cost of x. Then, the amortized cost for each node is $O(T^*)$.*

Proof Consider a node $x \in T$ and define $C(x) = \sum_{y \in N^+(x)} T(y)$. We divide the cost $T(x)$ proportionally among the children, so that each $y \in N^+(x)$ receives $T(x)\frac{T(y)}{C(x)}$. Observe that $\sum_{y \in N^+(x)} T(x)\frac{T(y)}{C(x)} = T(x)$, i.e. all the cost $T(x)$ is divided among the children. By doing this division recursively, starting from the root, we have that a node $z \in T$ receives from its parent at most $\frac{T(z)}{\alpha - 1}$.

Let us prove the last claim by induction on the depth of the node. Assume that all nodes w with depth $d - 1$ receive at most $\frac{T(w)}{\alpha - 1}$ from its parent. The base case is trivial, because the root receives no cost. Let z be a node with depth d, its parent w has depth $d - 1$. By the induction hypothesis, the cost received by w from its parent is $\frac{T(w)}{\alpha - 1}$ and the total cost of w is $T(w) + \frac{T(w)}{\alpha - 1}$. Thus, the cost received by z is:

$$\frac{T(z)}{C(w)}\left(T(w)+\frac{T(w)}{\alpha-1}\right)=T(z)\frac{T(w)}{C(w)}\frac{\alpha}{\alpha-1}\leq T(z)\frac{1}{\alpha}\frac{\alpha}{\alpha-1}=\frac{T(z)}{\alpha-1}.$$

The inequality follow from $\frac{T(w)}{C(w)}=\frac{T(w)}{\sum_{y\in N^+(w)}T(y)}\leq\frac{1}{\alpha}$. Completing the proof of the claim.

In the end of the cost division process the only nodes that have non-zero cost are the leaves. For any leaf the cost received from its parent is at most $\frac{T^*}{\alpha-1}$. Therefore, the total cost is $O(T^*+\frac{T^*}{\alpha-1})=O(T^*)$, since α is a constant.

Theorem 2.3 is a generalization of Lemma 2.3. In the theorem, for every node x satisfying item 2 we can use the same amortization strategy of the lemma, i.e. proportionally divide all the cost $T(x)$ among the children. However, instead of stopping this process only on the leaves, we stop on the first node satisfying item 1 or 3. If the node x satisfies item 1 and its ancestrals satisfy item 2, we know that the cost pushed to x is $O(\frac{T^*}{\alpha-1})$ and the total cost of x is $O(T^*+\frac{T^*}{\alpha-1})=O(T^*)$. On the other hand, if a node x satisfies item 3 we can amortize $T(x)$ among the $\Omega(\frac{T(x)}{T^*})$ children or solutions. In this way, each child receives $O(T^*)$ (that is not passed to its grandchildren), so that the cost of x is $O(T^*)$ and we have the same case of item 1.

Theorem 2.3 *Let T be a recursion tree, such that each node $x\in T$ satisfy one of the following properties:*

1. $T(x)=O(T^*)$;
2. $\sum_{y\in N^+(x)}T(y)\geq\alpha T(x)$, *where $\alpha>1$ is a constant;*
3. x *has $\Omega(\frac{T(x)}{T^*})$ children, or outputs $\Omega(\frac{T(x)}{T^*})$ solutions.*

Then, the amortized cost for each node is $O(T^*)$.

2.4.3.1 Matching Enumeration

Given an undirected graph $G=(V,E)$, a *matching* M in G is a set of pairwise non-adjacent edges, i.e. for any two edges $(u,v)\neq(x,y)\in M$, we have that $\{u,v\}\cap\{x,y\}=\emptyset$. The set of all matchings of G is denoted by $\mathscr{M}(G)$. Several variants of matching enumeration have been studied: perfect (every vertex has an incident edge in M) matching enumeration in bipartite graphs [26, 43–45], perfect matching in general graphs [46], maximal matchings in bipartite graphs [45] and maximal matchings in general graphs [47]. In this section, we consider the problem of enumerating all matchings of a graph. First, we present a simple algorithm (Algorithm 13) that correctly enumerates all matchings. Then, we modify it (Algorithm 14) to satisfy the hypothesis of Theorem 2.3, so we can use it to improve the algorithm complexity.

Algorithm 13 uses the binary partition method, each recursive call partitions $\mathscr{M}(G)$ in two sets: matchings not including the edge (u, v) and the ones including it. In the first case (line 6), we remove (u, v) from G and leave M unchanged. In the second case (line 7), we add (u, v) to the matching M and remove from G all edges adjacent to (u, v). The cost of a node is bounded by, the number of edges removed, $O(|V|)$. Now let us analyse the structure of the recursion tree. The conditional of line 1 ensures that only leaves output solutions. Moreover, there is always a matching including (u, v) and one not including it, so every node leads to a solution. Thus, in the recursion tree all internal nodes have exactly two children and every leaf output a solution. Therefore, the number of nodes is bounded by $2|\mathscr{M}(G)|$, and Algorithm 13 takes $O(|V|)$ time for each matching.

Algorithm 13: EnumMatching($G = (V, E), M$)

Input: A graph G and a matching M (eventually empty)
Output: $\mathscr{M}(G)$, the set of matchings of G

```
1 if E = ∅ then
2 |   output M
3 |   return
4 end
5 choose an edge (u, v) ∈ E
6 EnumMatching(G − (u, v), M)
7 EnumMatching(G − {(x, y) ∈ E|x ∈ {u, v}}, M ∪ {(u, v)})
```

Actually, each node in the recursion tree of Algorithm 13 takes $O(|N(u)|+|N(v)|)$ time. Consider a node x with the input graph $G = (V, E)$, the input graph of its children contain $|E| - 1$ and $|E| - |N(u)| - |N(v)|$ edges. Hereafter, for the sake of clear analysis, we bound the computation time of each node by $c|E|$. In this way, the cost for x is $T(x) = c|E|$ and, $T(y_1) = c(|E| - 1)$ and $T(y_2) = c(|E| - |N(u)| - |N(v)|)$ for each child. Based on this costs we cannot apply Theorem 2.3. The leaves take $O(1)$ time, satisfying item 1. However, the internal nodes do not satisfy item 2 or 3. Each internal node has exactly two children and do not output solutions, so that item 3 is not satisfied. On the other hand, the total computation time of the children is not increasing by constant factor over the parent, i.e. there is no constant $\alpha > 1$ such that $T(y_1) + T(y_2) \geq \alpha T(x)$.

In order to satisfy item 3 of Theorem 2.3 we need $|N(u)| + |N(v)|$ to be bounded, so that $T(y_2)$ is not too small. The key property is that for any graph either there is an edge (u, v) such that $|N(u)| + |N(v)| < |E|/2$ or there is a vertex u with $|N(u)| \geq |E|/4$. Algorithm 14 is a modified version of Algorithm 13 that uses this property. If there exists an edge (u, v) such that $|N(u)| + |N(v)| < |E|/2$ (line 5), we have that

$$T(y_1) + T(y_2) \geq c(|E| - 1) + c\frac{|E|}{2} \geq \frac{3}{2}T(x),$$

satisfying item 2. Alternatively, if there exists u such that $|N(u)| \geq |E|/4$ (line 8), we create at least $|E|/4$ children, satisfying item 3. Therefore, applying Theorem 2.3 and using that the number of internal nodes is bounded by $2|\mathscr{M}(G)|$, we have that Algorithm 14 takes $O(1)$ time amortized for each matching enumerated.

Algorithm 14: EnumMatching($G = (V, E)$, M)

Input: A graph G and a matching M (eventually empty)
Output: $\mathscr{M}(G)$, the set of matchings of G

```
1  if E = ∅ then
2  |  output M
3  |  return
4  end
5  if ∃(u, v) ∈ E s.t. |N(u)| + |N(v)| < |E|/2 then
6  |  EnumMatching(G − (u, v), M)
7  |  EnumMatching(G − {(x, y) ∈ E|x ∈ {u, v}}, M ∪ {(u, v)})
8  else
9  |  choose u s.t. |N(u)| ≥ |E|/4
10 |  EnumMatching(G − u, M)
11 |  foreach v ∈ N(u) do
12 |  |  EnumMatching(G − {(x, y) ∈ E|x ∈ {u, v}}, M ∪ {(u, v)})
13 |  end
14 end
```

2.5 Data-Driven Speed up

Polynomial delay algorithms, especially linear delay, can be considered as being very close to optimal algorithms in practice. Indeed since the delay is defined as a function of the input size, it can be often considered very small or negligible with respect to the usual exponential number of outputted solutions. However, there are certain applications, particularly in data mining, in which the input data are very large (frequent pattern mining, candidate enumeration, community mining, feasible solution enumeration). In such a context, when the number of outputted solutions is expected to be small (polynomial in the input size), the problem is said to be *large-scale tractable*. This is the case of enumerating peripheral or central vertices in large graphs, as shown in Chap. 7.

On the other hand, when the number of solutions increases exponentially with a linear increase in the instance size, we usually have that many solutions are similar and therefore redundant. In these cases, a post processing step is required to suppress the redundant solutions. Since the number of solutions is huge, the post processing is very time consuming or even infeasible, so these problems are considered intractable. This could be potentially the case of the frequent itemset problem, that is the problem of enumerating all the patterns appearing frequently in a large database, where a pattern can be a sequence of items, a short string, or a subgraph (any subset of a

frequent itemset is also frequent). However this intractability is very often linked to the application: for example, even if in theory the number of maximal cliques can be exponential in the size of the graph, in practice for large sparse graphs this number is usually polynomial in the size of the graph. On the other hand, however, the number of independent sets in a graph is huge both in theory and in practice.

Chapter 3
An Application: Biological Graph Analysis

3.1 Introduction

Biological networks, are currently being studied with approaches derived from the mathematical and physical sciences. Their structural analysis enables to highlight vertices or structures with special properties that have sometimes been correlated with the biological importance of a gene/protein or event. However, the way in which the biological networks are modelled often neglects the condition-dependencies and the relationship of their links, since a complex behaviour is forced in a form of representation of a simple graph, instead of a directed weighted hypergraph. Moreover, biological networks are dynamic both on the evolutionary time-scale, as well as on the much shorter time-scale of physiological processes. There is therefore not a unique network for a given cellular process, but potentially many realizations, each with different properties because of regulatory mechanisms. Such realizations provide snapshots of a same network in different conditions, enabling the study of condition-dependent structural properties. In such a context, by defining and solving a problem on this simplified form of representations, we need to check all its solutions in order to select the realistic ones.

Structure of the Chapter
The chapter is structured as follows: in Sect. 3.2 we report the overview of biological networks; in Sect. 3.3 we highlight the dynamical structure of the biological networks and we argue the importance of enumeration algorithms for biological network analysis. This second part of the chapter appeared in [2].

3.2 Biological Networks

High-throughput technologies have recently allowed a new perspective in biology, where the cell is interpreted as a large and complex system composed of highly integrated sub-systems. Interpretation of these systems as networks of interactions has

A. Marino, *Analysis and Enumeration*, Atlantis Studies in Computing 6,
DOI 10.2991/978-94-6239-097-3_3

spurred the application of analytical tools developed since long by mathematicians and physicists to biological networks.

Several kind of biological networks have been defined, such as Protein-Protein Interaction networks (see Sect. 3.2.1), Metabolic networks (see Sect. 3.2.2), Gene Regulatory networks (see Sect. 3.2.3), de Bruijn graph (see Sect. 3.2.4).

3.2.1 Protein-Protein Interaction Network

Proteins form an essential part of organisms and participate in virtually every process within cells. They can be enzymes that catalyse biochemical reactions, or they can have structural or mechanical functions. Moreover, they can be involved in cell signalling, immune responses, cell adhesion, and in the cell cycle.

Proteins are biochemical compounds consisting of one or more polypeptides. A polypeptide is a single linear polymer chain of amino acids bonded together by peptide bonds between the carboxyl (carbon double linked with oxygen) and amino (nitrogen linked to two hydrogen) groups of adjacent amino acid residues. The sequence of amino acids in a protein is defined by the sequence of a gene, which is encoded in the genetic code. In general, the genetic code specifies 20 standard amino acids. In the cell, a protein is produced by applying *transcription* and after *translation*.

Protein-protein interactions occur when two or more proteins bind together, often to carry out their biological function.

In a Protein-Protein Interaction network (in short PPIN), vertices are proteins and edges, that are undirected, represent physical interaction between them. Since these interactions can be further combined among them and can happen at different times, in a simple graph representation, whenever a protein has more than one partner (protein complex) we do not know if the different interactions take place together or at different times.

3.2.2 Metabolic Network

Metabolism is the set of chemical reactions that happen in the cells of living organisms to sustain life. These processes allow organisms to grow and reproduce, maintain their structures, and respond to their environments. A reaction transforms some chemical molecules into others.

The molecules that describe a reaction are called chemical compounds, or shortly compounds, and in particular, the chemical compounds involved in metabolism are called metabolites. The input compounds of a reaction are called substrates, while the output compounds are called products. Reactions may be reversible, meaning that it is possible to transform its set of products into its set of substrates.

A Metabolic network (in short MN) represents the set of chemical reactions involved in the metabolism of an organism, together performing tasks of putting together and breaking apart molecules in a living cell, e.g. photosynthesis, glycolysis. These chemical reactions are organized into metabolic pathways, in which one or more chemical are transformed through a series of steps into other chemicals, by a sequence of reactions. Enzymes allow organisms to drive desirable reactions that require energy and will not occur by themselves, by coupling them to spontaneous reactions that release energy.

A Metabolic network may be interpreted and built in different ways: vertices can be metabolites or reactions (respectively giving rise to the compound and the reaction graphs), and arcs can be reactions or shared metabolites (see [48, 49]). In particular, a Metabolic network can be modelled as a bipartite directed graph, whose set of vertices can be divided into a set of reactions $\mathscr{R}$ and a set of compounds $\mathscr{C}$, and whose set of arcs can be from a reaction to a compound and vice versa: a reaction has an incoming arc for each one of its substrates and one outgoing arc for each of its products. Alternatively a Metabolic network can be modelled as a directed hypergraph, whose vertices are compounds and hyperarcs are reactions: an hyperarc is a pair $(V_S(r), V_P(r))$, where $V_S(r)$ and $V_P(r)$ are respectively the set of substrates and the set of products of reaction r (see [50] for other examples of hypergraphs applied to biological questions and the associated computational problems).

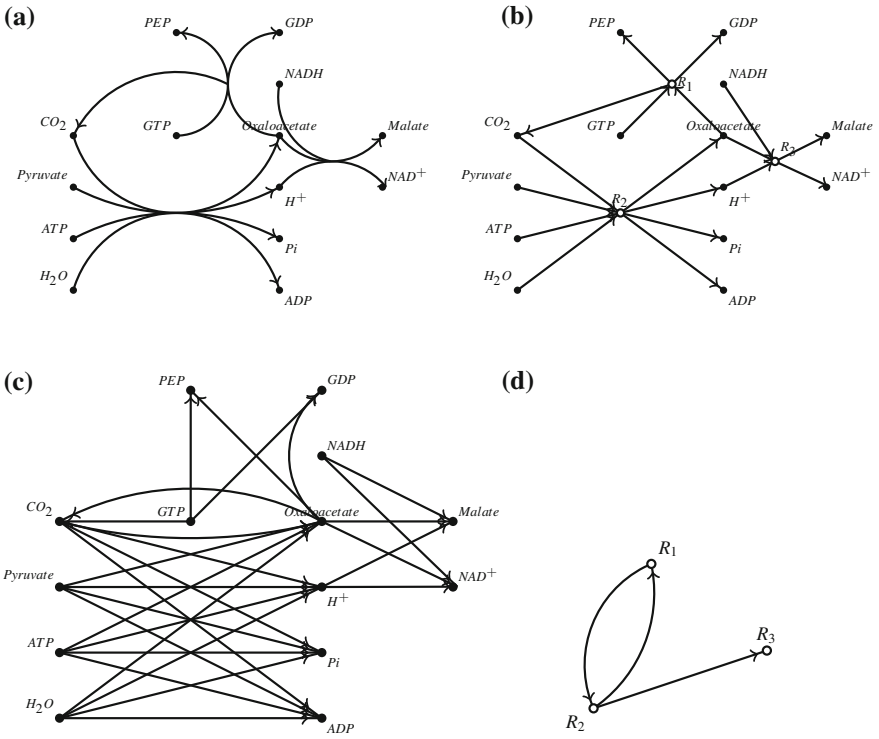
The reconstruction may lead to a loss of fundamental information, as for instance *stoichiometry*, that is the relative amount produced and consumed by each reaction. The stoichiometric matrix $S \in \mathbb{R}_{|C|\times|R|}$, is defined for any compound c and reaction r as follows.

$$S_{c,r} = \begin{cases} k & \text{if } r \text{ produces } k \text{ units of } c \\ -k & \text{if } r \text{ consumes } k \text{ units of } c \\ 0 & \text{otherwise} \end{cases}$$

Since Metabolic networks describe the reactions taking place inside a cell, there might be external compounds to the network: *input* (e.g. nutrients) and *output* (final product of a cell) compounds.

The problems modelled by using hypergraphs (or directed bipartite graphs) are usually hard, so that very often Metabolic networks are studied as Compound graphs, in which the vertices correspond to compounds and there is an arc between two compounds if there is a reaction where one is a substrate and the other is a product. For the sake of completeness, we will mention also Reaction graphs that are the graphs in which the vertices correspond to reactions and there is an arc between two reactions if there is a compound produced by one and consumed by the other [51].

In Fig. 3.1 we summarize these graph models to represent Metabolic networks by considering the reactions R_1, R_2, and R_3 involved in the *Gluconeogenesis* metabolic pathway.



$$R_1 : GTP + Oxaloacetate \longrightarrow CO_2 + PEP + GDP$$
$$R_2 : Pyruvate + CO_2 + ATP + H_2O \longrightarrow Oxaloacetate + H^+ + Pi + ADP$$
$$R_3 : NADH + Oxaloacetate + 2H^+ \longrightarrow Malate + NAD^+$$

Fig. 3.1 Modelling Metabolic networks. **a** Directed hypergraph. **b** Bipartite network. **c** Compound network. **d** Reaction network

3.2.3 Gene Regulatory Network

Regulation of gene expression (or gene regulation) includes the processes that cells and viruses use to regulate the way that the information in genes is turned into gene products. Although a functional gene product can be RNA, the majority of known mechanisms regulates protein coding genes. Gene regulation drives the processes of cellular differentiation and cell living cycle.

A Gene Regulatory network (GRN) models a collection of DNA segments in a cell which interact with each other indirectly, through their RNA and protein expression products, and with other substances in the cell, thereby governing the rates at which genes in the network are transcribed into mRNA.

In a Gene Regulatory network, signed arcs connect vertices representing transcriptional regulators to the vertices corresponding to their targets. The sign or weight

of the arcs indicates the effect of the control. Because of combinatorial regulation whose output depends on the architecture of promoters, which is not encoded in a basic Gene Regulatory network, an hypergraph representation could also be better for these networks [52–54].

Moreover, the system can have as input proteins such as transcription factors, and as output the level of gene expression. The dependencies among links can be modelled using new vertices that correspond to functions that can be obtained by combining basic functions upon the inputs (in a Boolean network model these are boolean functions, typically AND, OR, and NOT). These functions have been interpreted as performing a kind of information processing within the cell, which determines the cellular behaviour.

In Fig. 3.2 we summarize the main forms of representation of a Gene Regulatory network. A genetic circuit is a visual representation of a biological system. The bipartite graph has vertices for proteins and different logical gates for combinatorial regulation: AND requires the presence of both regulators to have transcription, while OR means that it can be activated by one of the regulators alone. The information on the promoter logic is lost in a simple representation, while it is encoded in a hypergraph representation. If a regulator z is removed, when analysing a simple network, one may infer that the auto-regulation of w continues to take place, which is not true, as correctly predicted by the bipartite graph.

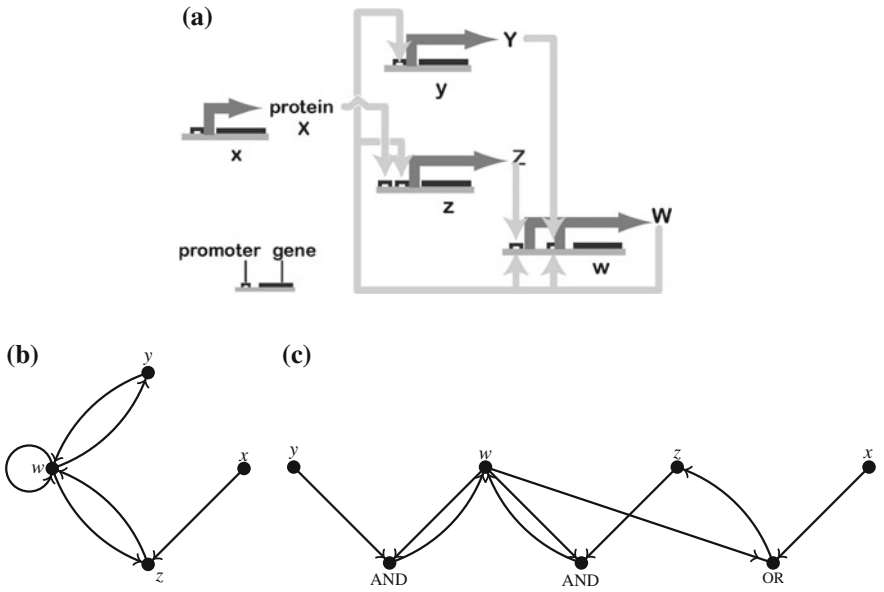


Fig. 3.2 Modelling Gene Regulatory networks. **a** Genetic circuit: *dark grey arrows* for transcription and translation, and *light grey arrows* for transcriptional regulation. **b** Simple network. **c** Bipartite network

3.2.4 De Bruijn Graph

Ever since Watson and Crick elucidated the structure of the DNA molecule in 1953, thus proving that it carried the genetic information, the challenge of reading the DNA sequence became central for biological research. The earliest chemical methods for DNA sequencing were extremely inefficient, laborious and costly. Over the next few decades, sequencing became more efficient by orders of magnitude. In the 1970s, two classical methods for sequencing DNA fragments were developed by Sanger and Gilbert, the dominant sequencing technologies from the late 1970s to the late 2000s used for all of the initial genome sequencing projects (*H. influenzae*, *Yeast*, *Drosophila*, *Arabidopsis*, *human*, and so on). In the 1980s these methods were augmented by the advent of partial automation as well as the cloning method.

Over the past couple of years, new sequencing technologies, called Next Generation Sequencing, have emerged. These new techniques sequence millions of fragments efficiently and in parallel. These fragment sequences are called *reads*, and they form the input for the computational problems.

Next generation sequencing can be used for SNP (single nucleotide polymorphism, i.e. a variation of a single nucleotide in the genetic code of a population) detection or even whole genome re-sequencing. In the first case, it requires the knowledge of most of the sequence in order to identify just rare differences among individuals. This can be used to model organisms that already have a high-quality reference genome sequence available. There are three next generation sequencing platforms that are commercially available and in widespread use: 454 (also known as pyrosequencing or Roche GS FLX, the first next generation method to be commercially available and the first to be applied to large-scale sequencing projects, such as sequencing the genome of James Watson), Solexa (also known as Illumina, used to sequence the entire genome of one African and one Asian human, plus the genome of a cancer patient), SOLiD (ABI).

Since all of these methods give short reads, they have mainly been used for resequencing. In this case, it is not necessary to do a complete, independent genome assembly, but the sequence reads can be aligned to a reference genome sequence. For example, the sequence reads from a single person can be aligned to the reference human genome. However, all of the above methods have been modified to produce *paired reads* in which both ends of a DNA fragment of known length are sequenced. This makes it possible to do *de novo* assemblies of genomes [55].

Performing a completely independent genome assembly is still an interesting topic of research: given h l-long reads $S_1, \ldots, S_h$, how to find the sequence of the full genome? In order to answer to this question the de Bruijn graph has been used. A de Bruijn graph (DBG) is a directed graph $G = (V, E)$ whose set of vertices V are labelled by k-mers, i.e. words of length k that are subsequences of the reads. An arc in E links a vertex u to a vertex v if the suffix of length $k - 1$ of u is a prefix of v. A path in this graph defines a potential contiguous subsequence in the genome sufficiently covered by the reads. Hence, a read can be converted to its corresponding path and the full genome sequence is a *long* walk in this graph.

3.3 Analysis and Enumeration of Biological Networks

Network metrics have been mainly developed for non-biological purposes, but in some cases they provided meaningful biological information. Several topological metrics are often used to analyse biological networks, like for instance, centrality to predict essential genes/reactions/compounds/proteins, average distance and diameter to inspect the compactness of a network, assortativity and dyadicity to study the modularity of a network and correlations between the properties of the vertices.

However, structural analyses are not always able to take into account regulatory mechanisms: the activity of enzymes is often regulated by one or more effector metabolites but since the effector metabolite is not consumed, they are not encoded in a Metabolic network. This can have profound consequences because these regulations have important roles in stabilizing metabolic states and in generating complex and biologically important dynamic behaviours [56–58]. These effectors are moreover able to cross the boundaries between different biological levels, such as metabolism and gene regulation; therefore building integrated models taking these cross talks into account represents a major challenge in systems biology. Modelling efforts have demonstrated that none of the different biological layers is truly isolated [59–61] so that perturbations propagate between them, and that enzymes also have regulatory functions, exerted through their control over the concentration of particular metabolites. These considerations lead to a view of the cell as a network of networks, whose understanding requires considering regulatory interactions not only within but also between biological networks.

Moreover, biases in the network reconstructions or manipulation can strongly affect the results of the analysis, confounding (if there exist) the correlations of biological and topological properties [62]. Indeed, topological measures are the result of a partial reconstruction and many measures are strongly affected by the sampling [63, 64].

In general, biological networks are often studied as static entities, but it should be stressed that they are instead very dynamic at widely different time-scales. They are dynamic in evolutionary time like any other biological structure, and even more on short time-scales, since regulatory connections and feedbacks change the connectivity of the network depending on the physiological state. Consequently, we should interpret most of the currently available biological network reconstructions as potential networks, where all the possible connections are indicated. By the term potential, we highlight the fact that edges/arcs and vertices in this network will be hardly present all together in vivo. If we consider for instance a Protein-Protein Interaction network, not all interaction partners of a protein will be expressed in a given condition, reducing the number of actual partners. On the converse, we may speak of network realizations when focusing on the active subgraph of the potential network, defined on the basis of experimental data [65–68]. The dynamic nature of biological networks is also at the basis of differential network analysis [69], which aims at capturing the subgraphs specific of a given network realization.

These considerations are important since they affect the analysis of biological networks. As there are many condition-specific realizations of a biological network, they plausibly have different structural properties. It has indeed been shown that random subgraphs of a network do not necessarily maintain the same degree distribution as the entire network [70], suggesting that other structural properties may also change. Similarly, it has been shown that power-law degree distributions can be obtained through random sampling of networks with different topology, indicating that it might be not possible to infer the true degree distribution from partial network reconstruction [71].

Several works try to take into account realizations. Han et al. [65] estimated the temporal connectivity of hubs in the *Yeast* Protein-Protein Interaction network by using gene expression data. Luscombe et al. [66] analysed the structural properties of the *Yeast* Gene Regulatory network in different conditions. Starting from a widely validated Gene Regulatory network, they used gene expression data to extract the subnetworks supposed to be active during environmental stress or the cell cycle, highlighting important differences (see also [65, 67, 68]). The use of realization networks is currently limited by the need for high-quality and high-throughput experimental data, today available only for a few organisms. Nevertheless, large-scale experimental data will be more easily obtained in the future, giving the occasion to develop the algorithms required for a similar approach.

Since biological networks are so complex, the structural analysis must take into account more biological information, whenever this is possible, or it requires enumerating all the feasible structures (given some set of constraints) in order to select a posteriori the realistic ones from a biological point of view. Thus, in this scenario, very often enumeration algorithms can be helpful and they have been applied for several purposes, for instance: enumerating interesting vertices [72], central or peripheral vertices (see Chap. 7), enumerating motifs [73–75], that are statistically overrepresented subgraphs in a network and have been recognized as "the simple building blocks of complex networks", and enumerating subgraphs [76, 77] (see Chap. 4), enumerating paths or cycles as chains of interactions or feedback loops [12] (see Chaps. 5 and 6), enumerating functional clusters [78], dense modules [79], or cliques [80].

Part II
Three Examples of Enumeration Algorithms

Chapter 4
Telling Stories: Enumerating Maximal Directed Acyclic Graphs with Constrained Set of Sources and Targets

4.1 Introduction

A classical goal of metabolic studies is to try to understand which are the metabolic processes involved in the adaptation to an environmental change. Although it is possible to keep track of some monitored metabolites, the internal mechanisms that lead to the observed variation are not clear. For "genome scale" networks, the metabolism of a whole organism is taken into account, while a metabolic perturbation may impact only a small portion of this complex network.

Recently, metabolomic techniques gained the spotlight by providing a way to monitor metabolism by measuring the concentration of metabolites in different conditions or time points. Typical results from these experiments are lists of metabolites whose concentrations significantly changed when the cell was exposed to stress. How to interpret this list of metabolites became then a new research topic, consisting in identifying the metabolic processes that link the metabolites of interest, possibly explaining the observed variations in their concentrations.

Some examples of approaches to deal with this kind of data may be found in [76, 81, 82].

Informally, we call a set of metabolic reactions linking all the metabolites of interest a *metabolic story*. For instance, a metabolomics study analysis compared a *Yeast* cell under two conditions, with and without exposition to cadmium [3]. The Metabolic network reconstruction of *Yeast* has about 1300 metabolites and the experiment identified a list of 22 metabolites whose concentrations changed. Figure 4.1 presents the *Yeast* network, and the highlight vertices, i.e. the light grey and the dark grey vertices, correspond to the metabolites identified in the experiment. The light grey vertices are those whose concentrations increased while the dark grey ones are the ones whose concentrations decreased. The concentrations of the other metabolites did not change significantly. Figure 4.2 presents a metabolic story that gives one possible explanation for the change in the concentration of the metabolites in the network. This particular story was found as one of the top scores using a score function that assigns a value to a story, after the enumeration process, based on the

A. Marino, *Analysis and Enumeration*, Atlantis Studies in Computing 6,
DOI 10.2991/978-94-6239-097-3_4

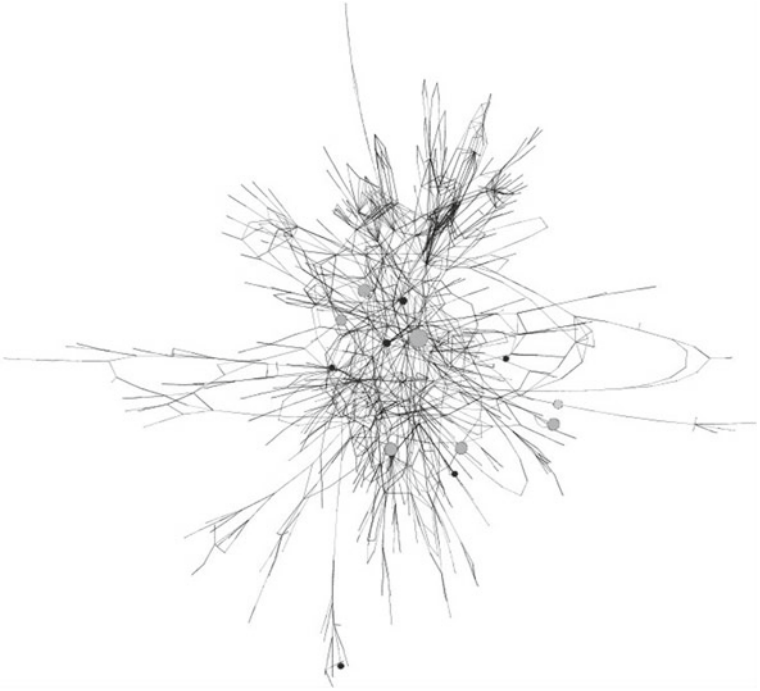


Fig. 4.1 Compound Metabolic Network of *Yeast*

concentration data from the metabolomic experiment. The chain of reactions present in this story is very close to the conclusions found in [3], since the order in which metabolites are transformed is almost the same. There are several other considered and finding all of them is the metabolic stories problem. Possible scenarios with a score close to the one of this story that could be considered and could provide new insights on the biology of the underlying metabolic process. Finding and enumerating all those alternatives is the metabolic stories problem.

Hence, a metabolic story should capture the relationship between the vertices of interest. Each individual story should explain how some metabolites are derived from others through a chain of reactions. In this sense, only light grey and dark grey vertices are allowed to be sources and targets in a story, even if they can appear as intermediate vertices in some stories. In order to enumerate all scenarios in which only dark grey or light grey vertices play the role of sources and/or targets, we introduce the acyclicity constraint. On the other hand, alternative pathways between these vertices that do not create cycles should always be included since they give additional explanations on the interconnection between them, and for this reason we introduce a maximality condition.

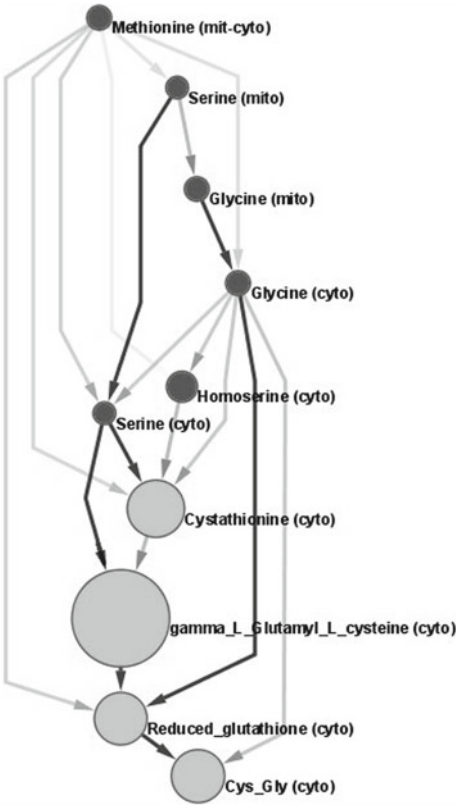


Fig. 4.2 A story of the Compound Metabolic Network of *Yeast*

Contribution

In this chapter we define the new problem of enumerating stories, that is a constrained version of the problem of enumerating all maximal directed acyclic subgraphs (DAG) of a graph G [6]. In our version, only a given subset $\mathbb{B}$ of the vertices are allowed to be sources or targets of the DAGs to be enumerated. For the computational problem, we will not distinguish dark and light grey metabolites, modelling them in the same way as black vertices.

The problem seems to be related to a Steiner tree/network problem, since the goal is to connect a distinguished set of vertices. Although the problem was originally motivated by biology where Steiner tree approaches have been widely explored [83], it is surprising that, as far as we know, such a constraint on sources and targets was never considered before edition. In this chapter, we show that introducing this constraint is enough to change the nature of the enumeration problem. Enumerating DAGs without the constraint is equivalent to enumerating feedback arc sets (FASs). A feedback arc set is a set of arcs that break all the cycles, i.e. it is the *complement* of a DAG. In this sense enumerating stories is a generalisation of enumerating minimal FASs, since the complement of a story is a minimal set of arcs that breaks all the

cycles and also avoids sources or targets that are not in $\mathbb{B}$. A sets of arcs that breaks all the cycles and also avoids sources or targets that are not in $\mathbb{B}$ is called *story arc sets* (SASs). Hence every SAS is a FAS. We show that not every minimal FAS is a minimal SAS, and give evidence that telling stories is possibly harder than enumerating minimal feedback arc sets.

In this chapter we will show a polynomial algorithm to compute one story, and then describe two different algorithms to enumerate all possible stories.

The contents of this chapter appeared in [4, 5, 84]. Moreover, the open problems we propose appeared in [7].

Structure of the Chapter

The chapter is organised in the following way. After introducing the main definitions and notations in Sect. 4.2, Sect. 4.3 presents some operations to simplify the graph without losing solutions. Section 4.4 shows a polynomial time algorithm for finding one story and also a proof that the problem of finding stories with a specific set of sources and targets is NP-complete. Sections 4.5.1 and 4.5.2 propose two different approaches to enumerate stories: the first one makes use of a minimal feedback-arc-set enumerator but can only be applied to a specific class of graphs while the second one is an extension of our algorithm to enumerate one story based on an initial permutation of the vertices and can be used for any graph; the example in Sect. 4.6 shows how this latter enumerator works. Section 4.7 provides complexity results for an alternative definition of stories and, finally, Sect. 4.8 conclude with some open problems.

4.2 Preliminaries

Let $G = (\mathbb{B} \cup \mathbb{W}, E)$ be a directed graph such that $\mathbb{B} \cap \mathbb{W} = \emptyset$. We write $V = \mathbb{B} \cup \mathbb{W}$. Vertices in $\mathbb{B}$ are said to be *black* while those in $\mathbb{W}$ are said to be *white*.

A *pitch* of G is an acyclic subgraph $G' = (\mathbb{B} \cup \mathbb{W}', E')$ of G with $\mathbb{W}' \subseteq \mathbb{W}$ and $E' \subseteq E$ and, for each vertex $w \in \mathbb{W}'$, $d^+(w) > 0$ and $d^-(w) > 0$. A trivial pitch is $G' = (\mathbb{B} \cup \emptyset, \emptyset)$: the subgraph containing all the black vertices and no arc. We define a *story* as a maximal pitch. We denote by $\Sigma(G)$ the set of stories of G. Thus, given $G = (\mathbb{B} \cup \mathbb{W}, E)$, we want to enumerate $\Sigma(G)$.

For independent reading, we define a *feedback arc set* (FAS) of a directed graph $G = (V, E)$, which is a subset F of E such that $G - F = (V, E \backslash F)$ is acyclic. A FAS is said to be *minimal* if there exists no $f \in F$ such that $F \backslash \{f\}$ is a FAS. We notice that, if $V = \mathbb{B} \cup \mathbb{W}$, the complement of a FAS is not always a story since $G - F$ may contain white sources or targets. Indeed, the FAS enumeration problem is a particular instance of our problem in which every vertex is black, i.e., $\mathbb{W} = \emptyset$. We define a *story arc set* (SAS) as a FAS S with the extra property that no white vertex in $G - S$ is a source or a target. A SAS is said to be *minimal* if there exists no subset S' of S such that $S \backslash S'$ is a SAS. This implies that if S is minimal, then for every $s \in S$, the graph $G = (\mathbb{B} \cup \mathbb{W}, (E \backslash S) \cup \{s\})$ either contains a cycle or

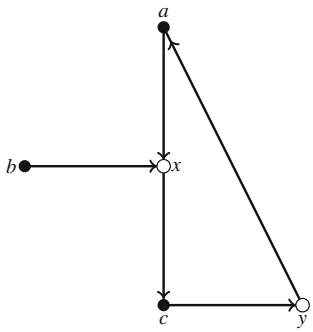


Fig. 4.3 In this case, $\mathbb{B} = \{a, b, c\}$ and $\mathbb{W} = \{x, y\}$. There are 4 possible minimal FASs: $\{(a, x)\}$, $\{(x, c)\}$, $\{(c, y)\}$, and $\{(y, a)\}$. Only one of these minimal FASs (that is, the first one) is also a minimal SAS. For example, the second one is not a SAS since $G - (x, c)$ contains a white target (that is, vertex x). On the other hand, another minimal SAS is $\{(c, y), (y, a)\}$, which is not a minimal FAS (even though it is a FAS)

contains a white source or target. If S is a minimal SAS, then $G - S$ is a story. A SAS is also a FAS. However, the example in Fig. 4.3 shows that, as expected, not every minimal FAS is a minimal SAS and, more surprisingly, that not every minimal SAS is a minimal FAS. For this reason, the use of a polynomial-time-delay enumeration algorithm for minimal FAS as the one proposed in [6] to enumerate stories is limited, since some minimal SASs may not be detected. We shall see in a later section that this is not the case when we restrict ourselves to a particular class of graphs.

4.3 Preprocessing the Graph

In this section, we show how a graph may be simplified without essentially changing the set of its stories. The simplifications allow shorter proofs of our results.

The simplified graphs turn out to be interesting from a biological point of view since they correspond to a more compact representation of graphs that is equivalent in terms of story sets. We applied the preprocessing steps described in this section on a collection of 107 Metabolic networks obtained from MetExplore [49, 85], randomly choosing sets of black vertices with sizes varying from 5 to 15 %. The compression ratio on the number of vertices goes from 65 to 98 % with an average reduction of 83 %, while the compression ratio on the number of arcs goes from 56 to 99 % with an average reduction of 77 %. This more compact representation of the interaction between the black vertices greatly facilitates the visualisation and analysis of the input data.

We now define the following four simplification operations:

- A *white source and target removal* consists in removing iteratively a white vertex from the graph that is either a source or a target. Clearly such vertices cannot appear in any story. Let $\texttt{de}(G)$ be the graph resulting of such removals.
- A *self-loop removal* consists in removing all arcs of the form (u, u). Since stories are acyclic, such arcs do not appear in any story. Let $\texttt{sl}(G)$ be the resulting graph of such removals.
- A *forward bottleneck removal* consists in removing a white vertex v whose out-degree is equal to 1, and directly connecting any predecessor of v to the unique successor of v (without creating multi-arcs). Let $\texttt{fb}(G, v)$ be the resulting graph.
- A *backward bottleneck removal* consists in removing a white vertex whose in-degree is equal to 1, and directly connecting the unique predecessor of v to the successors of v (without creating multi-arcs). Let $\texttt{bb}(G, v)$ be the resulting graph.

We prove that the last two operations leave the set of stories essentially unaltered. First an observation:

Observation 1 *Let v, p, and s be three vertices such that $(p, v), (v, s), (p, s) \in E$ and v is a (white) bottleneck. Then, for any story S, $(p, v), (v, s) \in S$ if and only if $(p, s) \in S$.*

Given three vertices $v, p, s \in V$ with $(p, v), (v, s) \in E$ and $(p, s) \notin E$, let $\texttt{ab}(G, v, p, s)$ denote the graph obtained by adding to G the arc (p, s).

Lemma 4.1 *Let $v \in \mathbb{W}$ be a forward bottleneck and let $p, s \in V$ be such that $(p, v), (v, s) \in E$ and $(p, s) \notin E$. Then there exists a bijection from $\Sigma(G)$ to $\Sigma(\texttt{ab}(G, v, p, s))$.*

Proof For any story $S \in \Sigma(G)$, we define $f(S) = S \cup \{(p, s)\}$ if $(p, v) \in S$ (and hence, $(v, s) \in S$ since v is a forward bottleneck), otherwise $f(S) = S$. To prove that $f(S) \in \Sigma(\texttt{ab}(G, v, p, s))$, we use Observation 1 to show that $f(S)$ is acyclic if and only if S is acyclic. We now show that $f(S)$ is maximal. Indeed, if $(p, s) \in f(S)$, then no set of arcs could be added to $f(S)$ since otherwise it could also be added to S. Otherwise, if (p, s) could be added to $f(S)$, then, from Observation 1 also (p, v) and (v, s) could be added to $f(S)$ and, hence, these two arcs could be added to S.

Let us now prove that, if S_1 and S_2 are two stories such that $S_1 \neq S_2$, then $f(S_1) \neq f(S_2)$. If $(p, v) \notin S_1 \cup S_2$, then $f(S_1) = S_1 \neq S_2 = f(S_2)$. Otherwise, if $(p, v) \in S_1 \cap S_2$, then $f(S_1) = S_1 \cup \{(p, s)\} \neq S_2 \cup \{(p, s)\} = f(S_2)$. Finally, if $(p, v) \in S_1 \backslash S_2$ (the other case can be dealt with similarly), then $(p, s) \in f(S_1)$ while $(p, s) \notin f(S_2)$ and, hence, $f(S_1) \neq f(S_2)$.

It remains to show that, for any $S' \in \Sigma(\texttt{ab}(G, v, p, s))$, there exists a $S \in \Sigma(G)$ such that $f(S) = S'$. Define $S = S' \backslash \{(p, s)\}$. Since S' is acyclic, so is S. If $(p, s) \notin S'$, then $S = S'$ and $S \in \Sigma(G)$, since the only difference between G and $\texttt{ab}(G, v, p, s)$ is the arc (p, s). Otherwise, from Observation 1, it follows that $(p, v), (v, s) \in S'$ and, hence, $(p, v), (v, s) \in S$: the maximality of S then follows from the maximality of S', since any set of arcs that could be added to S could also be added to S'.

By this lemma we may assume that, for any forward bottleneck $v \in \mathbb{W}$ whose unique successor is s, and for any predecessor p of v, the graph contains the arc (p, s). To complete the forward bottleneck removal operation, we then need to delete the vertex v without changing the stories set of the graph. Consider now the following operation: given a graph G with a forward bottleneck v, $\mathtt{dp}(G, v)$ denote the graph obtained by deleting from G the vertex v and all incident arcs.

Lemma 4.2 *Let $v \in \mathbb{W}$ be a forward bottleneck and s its unique successor. Suppose that for any predecessor p of v, the graph contains the arc (p, s). Then there is a bijection from $\Sigma(G)$ to $\Sigma(\mathtt{dp}(G, v))$.*

Proof For any $S \in \Sigma(G)$, we define $f(S) = S \backslash \{v\}$, that is the subgraph obtained by removing v and all incident arcs from S if $v \in S$. Since S is acyclic, so is $f(S)$. Moreover, from Observation 1, it follows that if $(p, v), (v, s) \in S$, then $(p, s) \in S$ and, hence, $(p, s) \in f(S)$. The maximality of $f(S)$ then follows from the maximality of S, since any set of arcs that could be added to $f(S)$ could also be added to S.

Let us now prove that, if S_1 and S_2 are two stories such that $S_1 \neq S_2$, then $f(S_1) \neq f(S_2)$. If $(p, s) \notin S_1 \cup S_2$, then $(p, v), (v, s) \notin S_1 \cup S_2$ and $f(S_1) = S_1 \neq S_2 = f(S_2)$. Otherwise, if $(p, s) \in S_1 \cap S_2$, then $(p, v), (v, s) \in S_1 \cap S_2$ and $f(S_1) = S_1 \backslash \{(p, v), (v, s)\} \neq S_2 \backslash \{(p, v), (v, s)\} = f(S_2)$. Finally, if $(p, s) \in S_1 \backslash S_2$ (the other case can be dealt with similarly), then $(p, s) \in f(S_1)$ while $(p, s) \notin f(S_2)$ and, hence, $f(S_1) \neq f(S_2)$.

Finally, let S' be a story of $\mathtt{dp}(G, v)$. Then S obtained by adding to S' the path $(p, v), (v, s)$ for every predecessor p of v such that $(p, s) \in S'$ is clearly a story and $f(S) = S'$.

Using the two previous lemmas, we obtain a justification for the third simplification operation.

Theorem 4.1 *For any forward bottleneck $v \in \mathbb{W}$, $\Sigma(G) = \Sigma(\mathtt{fb}(G, v))$.*

Analogously, we can justify the fourth operation.

Theorem 4.2 *For any backward bottleneck $v \in \mathbb{W}$, $\Sigma(G) = \Sigma(\mathtt{bb}(G, v))$.*

For any graph G, let $\mathtt{fb}(G)$ (respectively $\mathtt{bb}(G)$) denote the graph obtained by applying as many times as possible the forward (respectively backward) bottleneck removal operation. Notice that, even if G does not contain self-loops, it might happen that $\mathtt{fb}(G)$ (respectively $\mathtt{bb}(G)$) contains self-loops created by one bottleneck removal. Remember also that $\mathtt{sl}(G)$ denotes the graph obtained by the removal of all self-loops from G and $\mathtt{de}(G)$ denotes the graph obtained by the iterative removal of all white sources and targets from G. Our simplification procedure can now be described as follows.

(1) Let $G_0 = \mathtt{sl}(\mathtt{de}(G))$ and let $i = 0$.
(2) Let $G_{i+1} = \mathtt{sl}(\mathtt{bb}(\mathtt{sl}(\mathtt{fb}(G_i))))$.
(3) If $G_{i+1} = G_i$ then return G_i, otherwise let $i = i + 1$ and go to Step 2.

As a consequence of the previous results, we have that if H is the graph returned by this procedure, then there is a bijection between $\Sigma(G)$ and $\Sigma(H)$, and we may enumerate $\Sigma(H)$ instead. Hence from now on, we assume that any $v \in \mathbb{W}$ has $d^+(v) > 1$ and $d^-(v) > 1$. Notice that this avoids graphs like the one shown in Fig. 4.3. Indeed, in this case, the two arcs (c, y) and (y, a) would disappear and the arc (c, a) would be inserted. Furthermore, also x will disappear and we get arcs (b, c) and (a, c). Observe also that this simplification procedure does not guarantee that a minimal FAS enumerator would produce all possible minimal SAS as we shall see in the next section.

4.4 Finding Single Stories

Let us first consider the case of finding some story. We show that this can be done in polynomial time. Our algorithm basically starts with a pitch and grows it into a story by adding paths between black vertices while avoiding cycles. We can start with a trivial pitch such as the subgraph containing all the black vertices and no arcs.

Algorithm 15: COMPLETE_PITCH(G, P)

Input: A graph $G = (\mathbb{B} \cup \mathbb{W}, E))$ with $\mathbb{B} \cap \mathbb{W} = \emptyset$ and an initial pitch P;
Output: A story completing P

```
1  i ← 1;
2  π ← any topological order of P;
3  while i ≤ |V(P)| do
4  |  u ← i-th element according to π with u ∈ V(P);
5  |  Apply BFS(u, G\E(P)) until reach a vertex v ∈ V(P);
6  |  if π(u) < π(v) ∨ (u and v are incomparable) then
7  |  |  include the path u ⇝ v in P and update π;
8  |  |  i ← 1;
9  |  else
10 |  |  if no such vertex v exists then
11 |  |  |  i ← i + 1;
12 |  |  end
13 |  end
14 end
15 return P;
```

Theorem 4.3 *A story can be determined in polynomial time.*

Proof The algorithm COMPLETE_PITCH determines a story by completing a starting pitch P. It chooses a topological order π of the vertices consistent with the pitch. Starting in u, which can be any of the first vertices in this order that has not been scanned yet, a breadth-first search (BFS) is performed using only arcs not in $E(P)$. Any branch of the BFS tree is pruned as soon as it hits a vertex $v \in V(P)$. If v has

$\pi(u) < \pi(v)$ or u and v are incomparable, then the path $u \rightsquigarrow v$ is added to P and the topological order is updated. This addition creates no cycle since there was no path $v \rightsquigarrow u$ in P due to the fact that $\pi(u) < \pi(v)$ or u and v were incomparable, which can be checked in polynomial time. Moreover, since P contained no white source nor target before the addition of the path, then it does not contain any after adding the path because u and v, which are the only candidates to become source or target, were already present in P. Hence, the addition of $u \rightsquigarrow v$ to P creates a new pitch.

This procedure is repeated until no new path starting from u can be found. At this point, we continue with the next vertex in the updated order π. Every time a new path is found, π is updated and the procedure is started from the minimum vertex according to the new order. Since at each updating of the topological order, we add at least one arc, the algorithm terminates in polynomial time. The final pitch produced by this procedure is maximal and, therefore, a story.

We proceed by showing that the problem becomes NP-complete if we wish to identify a specific single story, i.e., one having a particular set of sources and/or targets.

Theorem 4.4 *Deciding whether there exists a story with a given set of sources and targets is NP-complete.*

Proof In order to prove this theorem, we show how the `3-SAT` problem [86] is reducible to the problem of deciding whether, given a directed graph $G = (V, E)$ and two subsets S and T of V, G contains a maximal DAG with its set of sources equal to S, and its set of targets equal to T. If this is true for maximal DAGs, it is also true for stories since any story is also a maximal DAG.

Consider a 3-CNF Boolean formula φ with clauses C_i, $i = 1, \ldots, m$, over a set Boolean variables x_j, $j = 1, \ldots, n$. We define a directed graph G as follows (see also Fig. 4.4).

- For each variable x_j, we create a set of six vertices, p_j, p_j^s, p_j^t, n_j, n_j^s, n_j^t, and for each clause C_i, two vertices s_i and t_i. We define the set $S = \{p_j^s, n_j^s \mid j =$

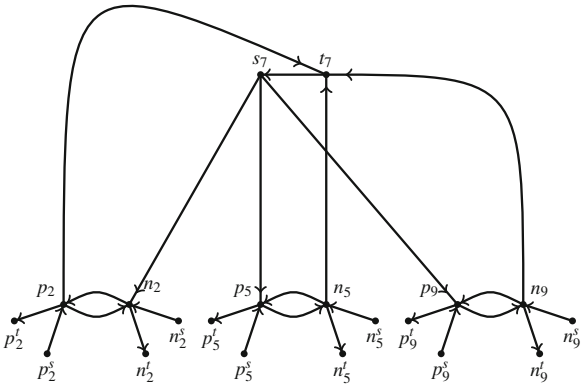


Fig. 4.4 The subgraph corresponding to the clause $C_7 = \neg x_2 \vee x_5 \vee x_9$

$1, \ldots, n\} \cup \{s_i \mid i = 1, \ldots, m\}$ and the set $T = \{p_j^t, n_j^t \mid j = 1, \ldots, n\} \cup \{t_i \mid i = 1, \ldots, m\}$.

- The set of arcs of G includes the six arcs

$$(p_j^s, p_j), (p_j, p_j^t), (p_j, n_j), (n_j, p_j), (n_j^s, n_j), (n_j, n_j^t)$$

related to each variable x_j and the arc (t_i, s_i) for each clause C_i.
- For each clause $C_i = l_i^1 \vee l_i^2 \vee l_i^3$, we introduce for each literal two arcs: if $l_i^h = x_j$ then we create the arcs (s_i, p_j) and (n_j, t_i), and if $l_i^h = \neg x_j$ the arcs (s_i, n_j) and (p_j, t_i), $h = 1, 2, 3$.

We prove that φ is satisfiable if and only if G includes a maximal DAG whose sets of sources and targets are, respectively, S and T.

Suppose φ is satisfiable and let τ be a satisfying truth-assignment. In the FAS F we include the arc (n_j, p_j) if $\tau(x_j) = \texttt{true}$ and the arc (p_j, n_j) if $\tau(x_j) = \texttt{false}$. Moreover, for each clause C_i, we include in F the arc (t_i, s_i) (see Fig. 4.5). Clearly, the resulting subgraph $G - F$ is a DAG whose set of sources (respectively, targets) is equal to S (respectively, T). Moreover, $G - F$ is maximal since removing any arc from F would create either a two-vertex variable cycle or, for some clause C_i, at least one six-vertex cycle corresponding to a true literal in C_i.

Now suppose that G' is a maximal DAG with sources S and targets T. Clearly, for each clause C_i, the arc (t_i, s_i) is not in G'. Maximality of G' implies that for each variable x_j, exactly one of (p_j, n_j) and (n_j, p_j) is in G'. All other arcs are included in G'. Let τ be a truth-assignment defined as follows: for each variable x_j, $\tau(x_j) = \texttt{true}$ if and only if (p_j, n_j) is in G'. We prove that this assignment satisfies φ. Suppose, to the contrary, that there exists an unsatisfied clause C_i. Wlog we may assume that $C_i = x_1 \vee x_2 \vee x_3$ (see Fig. 4.6). Then the three cycles containing the arc (t_i, s_i) are broken both by this arc and by the three arcs (n_j, p_j), $j = 1, 2, 3$ not in G'. Hence, G' is not maximal since the arc (t_i, s_i) can be added to G' without creating any new cycle. This contradicts the hypothesis on G'.

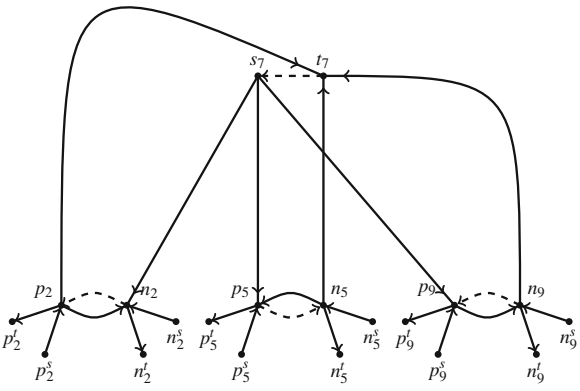


Fig. 4.5 The directed acyclic subgraph corresponding to the truth assignment $\tau(x_2) = \texttt{true}$, $\tau(x_5) = \texttt{false}$, and $\tau(x_9) = \texttt{true}$ that satisfies the clause $C_7 = \neg x_2 \vee x_5 \vee x_9$: the *dashed arcs* are in the FAS

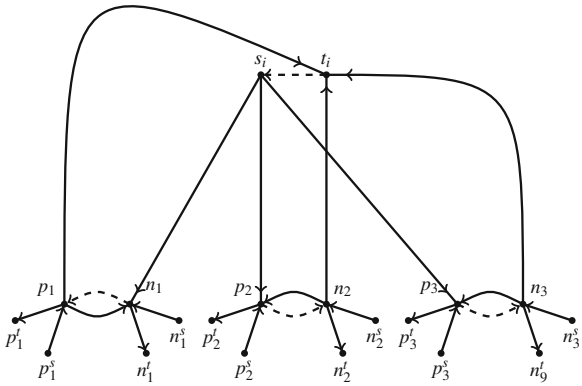


Fig. 4.6 A directed acyclic subgraph (the *dashed arcs* are in the FAS) corresponding to the truth assignment $\tau(x_2) = \texttt{true}$, $\tau(x_5) = \texttt{false}$, and $\tau(x_9) = \texttt{false}$ that does not satisfy the clause $C_7 = \neg x_2 \vee x_5 \vee x_9$: the DAG is not maximal since the arc (t_7, s_7) can be taken out from the FAS

It is easy to modify the previous reduction in order to prove that the same result holds even if we specify only the set of sources *or* only the set of targets.

4.5 Enumerating Stories

4.5.1 Enumerating Stories by Enumerating FAS*s*

We already noticed that there exist graphs for which the set $\mathscr{S}(G)$ of minimal SASs and the set $\mathscr{F}(G)$ of minimal FASs are not comparable in terms of the inclusion relation. In this section, we show that, for some particular cases, $\mathscr{S}(G)$ is contained in $\mathscr{F}(G)$.

A white vertex $v \in \mathbb{W}$ is called *bad* if, for any predecessor p of v and for any successor s of v, there exists a cycle containing the arcs (p, v) and (v, s) (see Fig. 4.7).

Proposition 4.1 *Any $v \in \mathbb{W}$, which is not bad, belongs to every story.*

Proof Consider a pitch P not containing v. As v is not bad, it has a predecessor p and a successor s such that there exists no cycle containing the arcs (p, v) and (v, s). By simplification rule 2, there exists a path $p_k, p_{k-1}, \ldots, p_1 = p$ with $k \geq 1$ such that $p_k \in \mathbb{B}$ and $p_i \in \mathbb{W}$, for any i with $i < k$. Let j be the minimum $i < k$ such that $p_i \in P$: if no such j exists, then we define $j = k$. Similarly a path $s = s_1, \ldots, s_{\ell-1}, s_\ell$ ending in a black vertex exists, and let $s_{j'}$ be the first vertex on that path belonging to P, or $s_{j'} = s_\ell$ if no such vertex exists.

Then $P' = P \cup \{(p_j, p_{j-1}), \ldots, (p, v), (v, s), \ldots, (s_{j'-1}, s_{j'})\}$ has no white source nor target as p_j and $s_{j'}$ are not white sources or targets in P. Moreover, P' is acyclic as P is acyclic and any cycle containing the additional path would contradict the fact that v is not a bad vertex. Thus any pitch not containing v is not maximal, hence not a story.

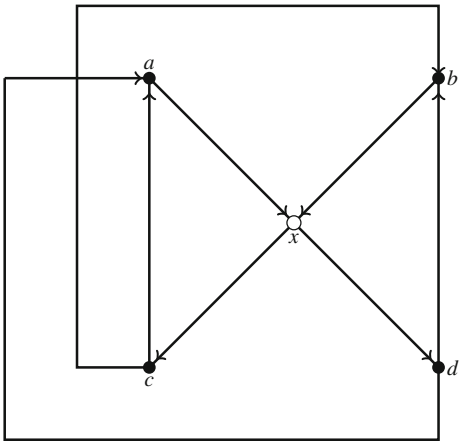


Fig. 4.7 Example of a bad vertex. The minimal SAS $\{(a, x), (b, x), (x, c), (x, d)\}$ is not a minimal FAS

Corollary 4.1 *If G does not include any bad vertex, then any minimal* SAS *is a minimal* FAS.

Proof By absurdum, assume that A is a minimal SAS which is not a minimal FAS. Then, there exists an arc $e = (u, v) \in A$ such that $A \backslash \{e\}$ is a FAS but not a SAS. This implies that in $G - (A \backslash \{e\})$, either u is a white target or v is a white source. We restrict ourselves to consider the latter case, since the former one can be dealt with similarly. Since v is a white source in $G - (A \backslash \{e\})$, and it is not in $G - A$, all arcs incident to v are in A. In other words, the story corresponding to A does not contain v, which contradicts Proposition 4.1.

The previous proposition and its corollary state that, in a graph with no bad vertices, each story corresponds to a minimal FAS. This suggests that for such graphs, we could enumerate all stories by enumerating all the minimal FASs and by checking for each of them whether the resulting graph is a story (which can be done by checking that no white vertex is source or target). Unfortunately, there are graphs with no bad vertices in which the number of minimal FASs is exponentially larger than the number of minimal SASs. An example is given in Fig. 4.8.

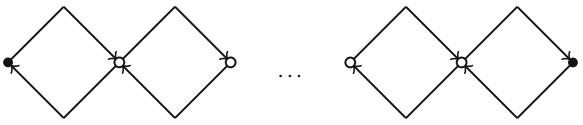


Fig. 4.8 Graph with no bad vertex and in which the number of minimal FASs is 2^n and the number of minimal SASs is 2

4.5.2 Enumerating Stories by Enumerating Permutations

In the previous section, we suggested a method for enumerating all stories in the case of graphs with no bad vertices. Unfortunately, many graphs arising from the biological application briefly described in Sect. 4.1 contain a huge number of bad vertices. We thus need a method for enumerating stories which is able to deal with these cases.

Remember how we can find a single story as explained in the proof of Theorem 4.3. Consider the following two simple operations, CLEAN and CONSISTENT_ARCS. For any graph $G(\mathbb{B} \cup \mathbb{W}, E)$, and for any total order π of the vertices:

- $G'(\mathbb{B} \cup \mathbb{W}, E') \equiv$ CONSISTENT_ARCS(G, π): for each arc $(u, v) \in E$, $(u, v) \in E'$ if $\pi(u) < \pi(v)$;
- $G'(\mathbb{B} \cup \mathbb{W}', E') \equiv$ CLEAN(G): recursively remove white vertices that are sources, targets or isolated in G.

We can thus define the composed operation

$$\text{PITCH}(G, \pi) = \text{CLEAN}(\text{CONSISTENT\_ARCS}(G, \pi)).$$

PITCH produces a pitch since the resulting graph G' contains only arcs that respect the order π and therefore is acyclic. Moreover, due to the cleaning step, G' is guaranteed to have neither white sources nor white targets.

Theorem 4.5 *For any story S, there exists a permutation π such that* PITCH$(G, \pi) = S$.

Proof It is enough to show that, for any story S of $G = (\mathbb{B} \cup \mathbb{W}, E)$ and for any topological order π of $V(S)$, PITCH$(G, \pi) = S$. Because of the maximality of a story, it suffices to show that $S \subseteq$ PITCH(G, π). Given an arc (u, v) of S, we have $\pi(u) < \pi(v)$. Therefore (u, v) is in CONSISTENT_ARCS(G, π). Since (u, v) is an arc of S, there exists a path p in S between two black vertices containing u and v. Then p is also in CONSISTENT_ARCS(G, π), and thus u and v are both black or, if one or both of them is white, then they are neither source nor target in CONSISTENT_ARCS(G, π). Since CLEAN(CONSISTENT_ARCS(G, π)) removes neither black nor white vertices that are neither source nor target, we conclude that (u, v) is also in CLEAN(CONSISTENT_ARCS(G, π)) = PITCH(G, π).

This theorem together with Theorem 4.3 suggest an approach to enumerate stories which simply consists in generating all permutations π of the vertices of G and computing $P =$ PITCH(G, π): if P is not a story, then we use COMPLETE_PITCH to grow it into a story.

In order to avoid to output several times the same story, we store in memory the previous solutions and every time we check whether the current story has been already generated.

It is worth observing that, in order to enumerate all the stories, all the possible permutations should be inspected. Thus the resulting total complexity is $\Omega(n!)$, even in the case of a graph with a constant number of stories, like the one shown in Fig. 4.8.

4.6 Enumerating Stories: An Example

Referring to the graph $G = (\mathbb{B} \cup \mathbb{W}, E)$ shown in Fig. 4.9a, the subgraph of G, $G' = (\mathbb{B} \cup \mathbb{W}, \{(a, b), (c, d), (e, f), (f, c)\})$ is a pitch. Let $\pi = \langle a, b, e, f, c, d \rangle$ be an ordering of its vertices. In order to complete this pitch in a story, Algorithm 15 consider in the first iteration the vertex b and a BFS from b until the vertices belonging to the pitch c and f are reached: the corresponding paths are simply the arcs out-going from b, (b, c) and (b, f); since b is not comparable with c and f, both the arcs are added. The next vertex to be considered is f: the BFS from f gives the arcs (f, a) and (f, d); the arc (f, a) cannot be added since $a > f$, while the arc (f, d) is added since $f < d$. The next vertex considered is thus d, and the BFS from d until the vertices belonging to the pitch are reached: the corresponding path is the arc (d, e) and, since $d > e$, this arc is not added to the pitch. The subgraph of G, shown in Fig. 4.9b, $G'' = (\mathbb{B} \cup \mathbb{W}, E'')$, where $E'' = \{(a, b), (b, c), (b, f), (c, d), (e, f), (f, c), (f, d)\}$, is thus a story.

Given a new ordering of the vertices $\pi = \langle f, d, e, a, b, c \rangle$, the operation CONSISTENT_ARCS activates the following set of arcs:

$$\{(a, b), (b, c), (d, e), (f, a), (f, c), (f, d)\}$$

that are the arcs compatible with the given order. Indeed the arcs $(b, f), (c, d), (e, f)$ are not compatible with π. Since f is a white source, the operation CLEAN deletes the arcs $(f, a), (f, c), (f, d)$. The resulting set of arcs $\{(a, b), (b, c), (d, e)\}$ induces a pitch. By applying Algorithm 15, it is thus possible to complete the pitch in order to get a story.

Observe that in the graph shown in Fig. 4.9a, it is possible to verify the Theorem 4.5: for any story S there exists an order π of its vertices such that $\text{PITCH}(G, \pi) = S$. Considering the story G'', shown in Fig. 4.9b, the ordering of

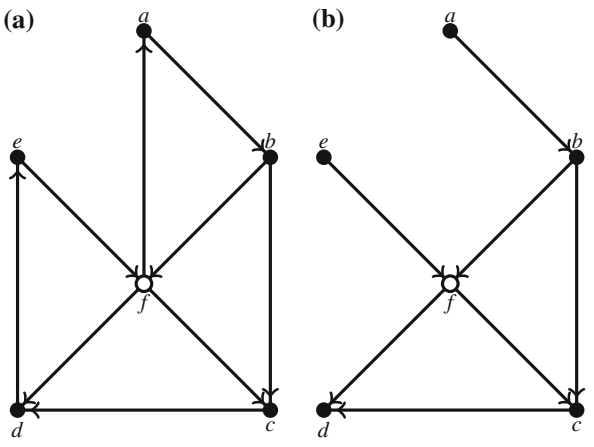


Fig. 4.9 An example of network and story

the vertices $\pi = \langle a, b, e, f, c, d \rangle$ is such that PITCH$(G, \pi) = G''$. Indeed by starting from this order, the operation CONSISTENT_ARCS activates all the arcs in E'', that are the arcs compatible with the given order. The arcs (d, e) and (f, a) are indeed not compatible with π. The operation CLEAN does not affect the solution, since no white source or target is present. Thus E'' induces a pitch that is also a story.

4.7 Alternative Definition of a Story

It is clear that, according to our definition of a story, no white vertex can be either source or target in the original graph, since otherwise such a white vertex would not belong to any story. This implies that the original graph can be seen as the union of a finite set $\mathscr{P}$ of directed paths between black vertices: in particular, if $\mathscr{P}$ includes all paths between every pair of black vertices, then it is easy to verify that a story is a maximal subset $\mathscr{S}$ of $\mathscr{P}$ such that the graph defined as the union of the paths in $\mathscr{S}$ is acyclic and there exists no path p in $\mathscr{P} - \mathscr{S}$ that can be added to $\mathscr{S}$ without disturbing the acyclicity. Let us call this alternative definition of story a *path-story*. A minimal number of paths to be removed from $\mathscr{P}$ such that the union of the remaining paths is a path-story is called a *feedback path set*.

A natural question is whether the problem changes when a set $\mathscr{P}$ is given as input, and the graph $G_{\mathscr{P}}$ is defined by the union of the paths of $\mathscr{P}$, where the endpoints of the paths in $\mathscr{P}$ form the set of black vertices of $G_{\mathscr{P}}$. Clearly, since $\mathscr{P}$ may not contain all paths between every pair of the black vertices in $G_{\mathscr{P}}$, the set of path-stories of $G_{\mathscr{P}}$ is different from the set of stories of $G_{\mathscr{P}}$ (see for an example Fig. 4.10). We will prove that enumerating path-stories is at least as hard as enumerating hitting sets, which is a well-known enumeration problem (for a survey, we refer to [87]) with its computational complexity still open, after more than 28 years.

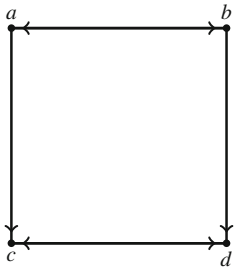


Fig. 4.10 Graph obtained by two paths (a, b, d, c) and (b, a, c, d). According to the alternative definition, this graph clearly contains only two stories, which correspond to the two paths. According to the original definition, instead, the graph contains the following four minimal SAS: $\{(a, b), (c, d)\}$, $\{(a, b), (d, c)\}$, $\{(b, a), (c, d)\}$, and $\{(b, a), (d, c)\}$. Note that these four minimal SAS originated four stories which are all different from the two stories obtained according to the second definition

Theorem 4.6 *Enumerating path-stories is at least as hard as enumerating minimal hitting sets.*

Proof Let $\mathscr{C}$ be a collection of subsets of a domain set X. $H \subset X$ is a *hitting set* of $\mathscr{C}$ if for any $C \in \mathscr{C}$, $H \cap C \neq \emptyset$.

We reduce $\mathscr{C}$ to a collection $\mathscr{P}$ of paths, such that there is a bijective correspondence between (minimal) hitting sets of $\mathscr{C}$ and (minimal) feedback path sets of $\mathscr{P}$ and, hence, between hitting sets of $\mathscr{C}$ and path-stories of $\mathscr{P}$.

We order all sets of $\mathscr{C}$ and all elements of X. Within any set of $\mathscr{C}$ the elements are ordered. For each element in each set we create a vertex of the graph $G_{\mathscr{P}}$. For each set $C_i \in \mathscr{C}$ with $C_i = \{x_{i_1}, \ldots, x_{i_{k_i}}\}$, create a cycle by introducing the arcs $(x_{i_\ell}, x_{i_{\ell+1}})$, $\ell = 1, \ldots, k_i$ and $(x_{i_{k_i}}, x_{i_1})$. We call this cycle also C_i. Moreover, suppose that $x_{i_\ell} = x_j$ is the h-th occurrence of x_j and x_{r_t} the next occurrence, then we introduce an arc $(x_{i_{\ell+1}}, x_{r_t})$, i.e., there is a path of two arcs between any two consecutive occurrences of the same element. Let us call the latter set of arcs the *element-arcs* and the set of arcs on the cycles the *set-arcs*. Notice that the element arcs are not in any cycle.

Now for each element x_j we define a path $P_j \in \mathscr{P}$, by starting in the vertex of the first occurrence of x_j, and every time selecting the two arcs connecting it to the next occurrence vertex, until we arrive at the last occurrence vertex.

The graph induced by $\mathscr{P}$ contains all the arcs just introduced. In particular it contains all the cycles corresponding to the sets in $\mathscr{C}$. An example of the reduction is shown in Fig. 4.11.

It is easy to see that a path P_j cuts cycle C_i if and only if x_j hits the set C_i. Hence there is a one-to-one correspondence between a minimal path set of $\mathscr{P}$ and a minimal hitting set of $\mathscr{C}$. This proves the theorem.

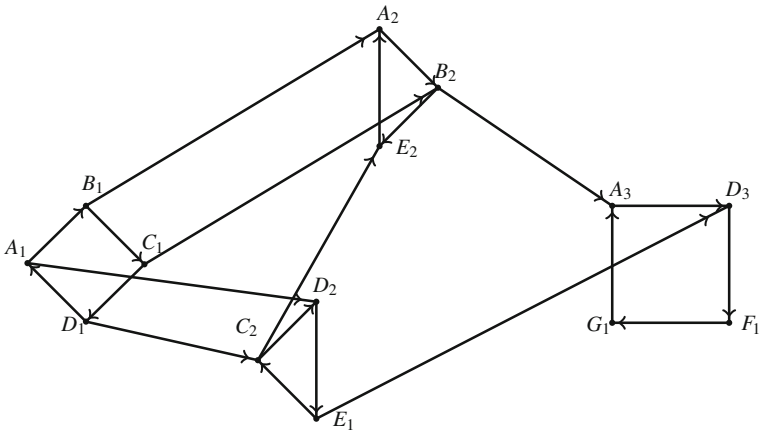


Fig. 4.11 An example of reduction: $C_1 = \{A, B, C, D\}$, $C_2 = \{C, D, E\}$, $C_3 = \{A, B, E\}$, and $C_4 = \{A, D, F, G\}$

In this chapter we have mainly focused our attention on the first definition of stories, since this definition seems to fit better with the informal subnetwork definition the biologists are looking for.

4.8 Conclusion and Open Problems

In this chapter, we have introduced the new notion of a story, which is a maximal acyclic subgraph of a directed graph in which only specified vertices can be sources or targets. We have proved some complexity results and designed some algorithms for enumerating all possible stories of a graph. From a theoretical point of view, the main question left open is to establish the complexity of the enumeration problem. Indeed the enumeration algorithm presented, even if it works well in practice, as shown by [84], gives no guarantee on the delay between the output of two consecutive solutions. Notice that any changes in the definition will imply a revision of the formal results, and may even imply that finding one story cannot be polynomial-solvable. We address as a future work, exploiting the relationship between stories and subset feedback vertex sets, that has been studied in [88] by applying a *Measure and Conquer* approach [89]. The algorithm presented here has been recently improved in [90] by enumerating pitches with linear time delay.

From a practical point of view, for some graphs the number of solutions found is extremely large and therefore the analysis of the results is compromised. Adding more constraints to the model could be a way to filter a priori the set of solutions.

This observation on the size of the output leads us to consider the problem from a modeling point of view. For instance, the acyclicity constraint could be relaxed allowing cycles between white vertices. Moreover, the model could be enriched by exploring the information on the concentrations given by the metabolomics experiment. Notice that in this case the nature of the problem changes into an optimization problem. Another alternative is to consider integrated models, adding to the Metabolic network other layers of information such as regulation, or taking the stoichiometry of the reactions into account. Finally, in metabolomics a current challenge is to correctly predict which are the metabolites corresponding to the peaks in the spectrum and whether the changes in concentration are actually significant, which suggests that the model should also account for noisy data corresponding to on incertitude on the *interesting* or *not interesting* labels assigned to the vertices.

Chapter 5
Enumerating Bubbles: Listing Pairs of Vertex Disjoint Paths

5.1 Introduction

In recent papers [8, 91], algorithms for identifying two types of polymorphism, respectively SNPs (Single Nucleotide Polymorphisms) in DNA, and alternative splicing in RNA-seq data were introduced. Both correspond to recognisable patterns in a de Bruijn graph (DBG) built from the reads provided by a sequencing project. In both cases, the pattern corresponds to two vertex-disjoint paths between a pair of source and target vertices s and t. Properties on the lengths or sequence similarity of the paths then enable to differentiate between different types of polymorphism.

Such patterns have been studied before in the context of genome assembly where they have been called bulges [92] or bubbles [93–95]. However, the purpose in these works was not to enumerate all these patterns, but "only" to remove them from the graph, in order to provide longer contigs for the genome assembly. More recently, ad-hoc enumeration methods have been proposed but are restricted to non-branching bubbles [96], i.e., each vertex from the bubble has in-degree and out-degree 1, except for s and t. Furthermore, in all these applications [92–96] since the patterns correspond to SNPs or sequencing errors, the authors only considered paths of length smaller than a constant. On the other hand, bubbles of arbitrary length have been considered in the context of splicing graphs [97]. However, in this context, a notable difference is that the graph is a DAG. Additionally, vertices are coloured and only unicolour paths are then considered for forming bubbles. Finally, the concept of bubble also applies to the area of phylogenetic networks [98], where it corresponds to the notion of a recombination cycle. Again for this application, the graph is a DAG.

In this chapter, we adopt the term bubble, which is being most used in the community, and this will denote two vertex-disjoint paths between a pair of source and target vertices with no condition on the path length or the degrees of the internal vertices. We then consider the more general problem of enumerating all bubbles in a arbitrary directed graph. That is, our solution is not restricted to acyclic or de Bruijn graphs. This problem is quite general but it was still an open question whether a polynomial-delay algorithm could be proposed for solving it. The algorithm presented in [8] was

A. Marino, *Analysis and Enumeration*, Atlantis Studies in Computing 6,
DOI 10.2991/978-94-6239-097-3_5

an adaptation of Tiernan's algorithm for cycle enumeration [9] which does not have a polynomial delay, in the worst case the time elapsed between the output of two solutions is proportional to the number of paths in the graph, i.e. exponential in the size of the graph. It was not clear at the time if more efficient cycle enumeration methods in directed graphs such as Tarjan's [28] or Johnson's [10] could be adapted to efficiently enumerate bubbles in directed graphs.

Contribution
The aim of this chapter is to show a non-trivial adaptation of Johnson's cycle (what he called elementary circuit) enumeration algorithm to identify all bubbles in a directed graph in the same theoretical complexity. Notably, the method we propose enumerates all bubbles with a given source with $O(|V| + |E|)$ delay. The algorithm requires an initial transformation, described in Sect. 5.3, of the graph for each source s that takes $O(|V| + |E|)$ time and space. Our work appeared in [11].

Structure of the Chapter
The chapter is organised as follows. We start by recalling in Sect. 5.2 what is a de Bruijn graph representation of a set of reads, and how polymorphisms in DNA- and RNA-seq data correspond to bubbles in this graph. We then explain in Sect. 5.3 how to transform the original graph into a new graph where bubbles will correspond to cycles with some properties. We present in Sect. 5.4 the algorithm for enumerating all cycles corresponding to bubbles in the initial graph and we provide an example in Sect. 5.5. We prove in Sect. 5.6 that this algorithm is correct and has linear delay; in Sect. 5.7 we explain how to avoid duplicate bubbles with no additional cost. Finally in Sect. 5.8 we conclude with some open problems.

5.2 Preliminaries

Recall that a de Bruijn graph (DBG) is a directed graph $G = (V, E)$ whose set of vertices V are labelled by k-mers, i.e. words of length k. An arc in E links a vertex u to a vertex v if the suffix of length $k - 1$ of u is a prefix of v. By an (s, t)-bubble, we mean two vertex-disjoint (s, t)-paths that only shares s and t.

In the case of next generation sequencing (NGS) data, the k-mers correspond to all words of length k present in the reads (strings) of the input dataset, and only those. In relation to the classical de Bruijn graph for all possible words of size k, the DBG for NGS data may then not be complete. Vertices may also be labelled by the number of times each k-mer is present in the reads. In general a vertex will be labelled by both a k-mer and its reverse complement, and the DBG used in practice will thus be a bi-directed multigraph. Figure 5.1 gives an example of a portion of a DBG that corresponds to a bubble generated by a SNP or a sequencing error.

In this chapter, we ignore all details related to the treatment of NGS data using de Bruijn graphs that are not essential for the algorithm described, and consider instead the more general case of finding all (s, t)-bubbles in an arbitrary directed graph.

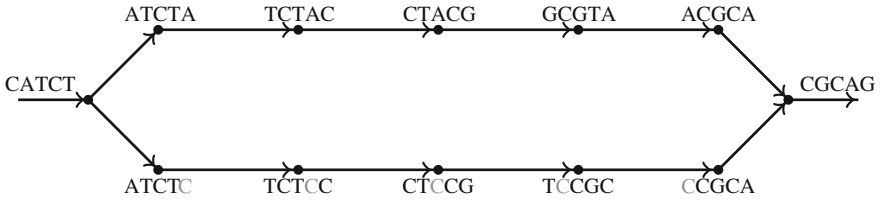


Fig. 5.1 Bubble due to a substitution (*gray letter*)

5.3 Turning Bubbles into Cycles

Let $G = (V, E)$ be a directed graph, and let $s \in V$. We want to find all (s, t)-bubbles for all possible target vertices t. We transform G into a new graph $G'_s = (V'_s, E'_s)$ where $|V'_s| = 2|V|$ and $|E'_s| = O(|V| + |E|)$. Namely,

$$V'_s = \{v, \overline{v} \mid v \in V\}$$

$$E'_s = \{(u, v), (\overline{v}, \overline{u}) \mid (u, v) \in E \text{ and } v \neq s\} \cup \{(v, \overline{v}) \mid v \in V \text{ and } v \neq s\} \cup \{(\overline{s}, s)\}$$

Let us denote by $\overline{V}$ the set of vertices of G'_s that were not already in G, that is $\overline{V} = V'_s \setminus V$. The two vertices $x \in V$ and $\overline{x} \in \overline{V}$ are said to be *twin vertices*. Observe that the graph G'_s is thus built by adding to G a reversed copy of itself, where the copy of each vertex is referred to as its *twin*. The arcs incoming to s (and outgoing from $\overline{s}$) are not included so that the only cycles in G'_s that contain s also contain $\overline{s}$. New arcs are also created between each pair of twins: the new arcs are the ones leading from a vertex u to its twin $\overline{u}$ for all u except for s where the arc goes from $\overline{s}$ to s. An example of a transformation is given in Fig. 5.2.

We define a cycle of G'_s as being *bipolar* if it contains vertices of both V and $\overline{V}$. As the only arc from $\overline{V}$ to V is $(\overline{s}, s)$, then every bipolar cycle C contains also only

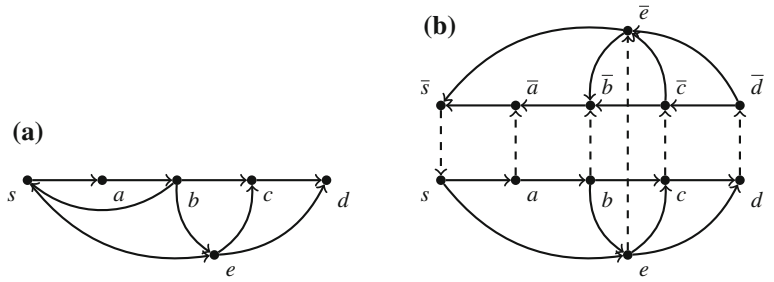


Fig. 5.2 Graph G and its transformation G'_s. We have that $\langle s, e, \overline{e}, \overline{b}, \overline{a}, \overline{s}, s\rangle$ is a bubble-cycle with swap arc $(e, \overline{e})$ that has a correspondence to the (s, e)-bubble composed by the two vertex-disjoint paths $\langle s, e\rangle$ and $\langle s, a, b, e\rangle$. **a** Graph G. **b** Graph G'_s

one arc from V to $\overline{V}$. This arc, which is the arc $(t, \bar{t})$ for some $t \in V$, is called the *swap arc* of C. Moreover, since $(\bar{s}, s)$ is the only incoming arc of s, all the cycles containing s are bipolar. We say that C is *twin-free* if it contains no pair of twins except for $(s, \bar{s})$ and $(t, \bar{t})$.

Definition 5.1 (*Bubble-cycle*) A *bubble-cycle* in G'_s is a twin-free cycle of size greater than four.[1]

Proposition 5.1 *Given a vertex s in G, there is a one-to-two correspondence between the set of (s, t)-bubbles in G for all $t \in V$, and the set of bubble-cycles of G'_s.*

Proof Let us consider an (s, t)-bubble in G formed by two vertex-disjoint (s, t)-paths P and Q. Consider the cycle of G'_s obtained by concatenating P (resp. Q), the arc $(t, \bar{t})$, the inverted copy of Q (resp. P), and the arc $(\bar{s}, s)$. Both cycles are bipolar, twin-free, and have $(t, \bar{t})$ as swap arc. Therefore both are bubble-cycles.

Conversely, consider any bubble-cycle C and let $(t, \bar{t})$ be its swap arc. C is composed by a first subpath P from s to t that traverses vertices of V and a second subpath $\overline{Q}$ from $\bar{t}$ to $\bar{s}$ composed of vertices of $\overline{V}$ only. By definition of G'_s, the arcs of the subpath P form a path from s to t in the original graph G; given that the vertices in the subpath $\overline{Q}$ from $\bar{t}$ to $\bar{s}$ are in $\overline{V}$ and use arcs that are those of E inverted, then Q corresponds to another path from s to t of the original graph G. As no internal vertex of $\overline{Q}$ is a twin of a vertex in P, these two paths from s to t are vertex-disjoint, and hence they form an (s, t)-bubble.

Notice that there is a cycle $s, v, \bar{v}, \bar{s}$ for each v in the out-neighbourhood of s. Such cycles do not correspond to any bubble in G, and the condition on the size of C allows us to rule them out.

5.4 The Algorithm

In the previous chapter we have seen several techniques to enumerate cycles, and in particular the Johnson's Algorithm [10], that is a polynomial delay algorithm for the cycle enumeration problem that works also in the case of directed graphs. We propose to adapt the principle of this latter algorithm because, since the graphs in which we are interested in are directed, we cannot apply the algorithm presented in Chap. 6.

In particular we will use the idea of the pruned backtracking, to enumerate bubble-cycles in G'_s, modified to take into account the twin vertices. Proposition 5.1 then ensures that running our algorithm on G'_s for every $s \in V$ is equivalent to the enumeration of (twice) all the bubbles of G. To do so, we explore G'_s by recursively traversing it while maintaining the following three variables. We denote by $N^+(v)$ the set of out-neighbours and $N^-(v)$ as the set of in-neighbours of v.

[1] The only twin-free cycles in of size four in G'_s are generated by the outgoing arcs of s. There are $O(|V|)$ of such cycles.

1. A variable *stack* which contains the vertices of a path (with no repeated vertices) from s to the current vertex. Each time it is possible to reach $\overline{s}$ from the current vertex by satisfying all the conditions to have a bubble-cycle, this stack is completed into a bubble-cycle and its content output.
2. A variable *status*(v) for each vertex v which can take three possible values:
   - *free*: v should be explored during the traversal of G'_s;
   - *blocked*: v should not be explored because it is already in the stack or because it is not possible to complete the current stack into a cycle by going through v—notice that the key idea of the algorithm is that a vertex may be blocked without being on the stack, avoiding thus useless explorations;
   - *twinned*: $v \in \overline{V}$ and its twin is already in the stack, so that v should not be explored.
3. A set $B(v)$ of in-neighbours of v where vertex v is blocked and for each vertex $w \in B(v)$ there exists an arc (w, v) in G'_s (that is, $w \in N^-(v)$). If a modification in the stack causes that v is unblocked and it is possible to go from v to $\bar{s}$ using free vertices, then w should be unblocked if it is currently blocked.

Algorithm 16 enumerates all the bubble-cycles in G by fixing the source s of the (s, t)-bubble, computing the transformed graph G'_s and then listing all bubble-cycles with source s in G'_s. This procedure is repeated for each vertex $s \in V$. To list the bubble-cycles with source s, procedure CYCLE(s) is called. As a general approach, Algorithm 18 uses classical backtracking with a pruned search tree. The root of the recursion corresponds to the enumeration of all bubble-cycles in G'_s with starting point s. The algorithm then proceeds recursively: for each free out-neighbour w of v the algorithm enumerates all bubble-cycles that have the vertices in the current stack plus w as a prefix. If $v \in V$ and $\overline{v}$ is twinned, the recursion is also applied to the current stack plus $\overline{v}$, $(v, \overline{v})$ becoming the current swap arc. A base case of the recursion happens when $\overline{s}$ is reached and the call to CYCLE($\overline{s}$) completed. In this case, the path in *stack* is a twin-free cycle and, if this cycle has more than 4 vertices, it is a bubble-cycle to output.

The key idea that enables to make this pruned backtracking efficient is the block-unblock strategy. Observe that when CYCLE(v) is called, v is pushed in the stack and to ensure twin-free extensions, v is blocked and $\overline{v}$ is twinned if $v \in V$. Later, when backtracking, v is popped from the stack but it is *not necessarily* marked as free. If there were no twin-free cycles with the vertices in the current stack as a prefix, the vertex v would remain blocked and its status would be set to free only at a later stage. The intuition is that either v is a dead-end or there remain vertices in the stack that block all twin-free paths from v to $\overline{s}$. In order to manage the status of the vertices, the sets $B(w)$ are used. When a vertex v remains blocked while backtracking, it implies that every out-neighbour w of v has been previously blocked or twinned. To indicate that each out-neighbour $w \in N^+(v)$ (also, $v \in N^-(w)$ is an *in-neighbour* of w) blocks vertex v, we add v to each $B(w)$. When, at a later point in the recursion, a vertex $w \in N^+(v)$ becomes unblocked, v must also be unblocked as

Algorithm 16: Main algorithm

```
Input: A graph G = (V, E)
Output: All the bubbles in G
1 for s ∈ V do
2 |   stack ← ∅;
3 |   for v ∈ G'_s do
4 |   |   status ← free;
5 |   |   B(v) = ∅;
6 |   end
7 |   CYCLE(s);
8 end
```

Algorithm 17: Procedure *UNBLOCK*(v)

```
Input: A vertex v ∈ V
/* recursive unblocking of vertices for which popping v
   creates a path to s̄                                    */
1 status(v) ← free;
2 for w ∈ B(v) do
3 |   delete w from B(v);
4 |   if status(w)= blocked then
5 |   |   UNBLOCK(w);
6 |   end
7 end
```

possibly there are now bubble-cycles that include v. Algorithm 17 implements this recursive unblocking strategy.

An important difference between the algorithm introduced here and Johnson's is that we now have three possible states for any vertex, i.e. free, blocked and twinned, instead of only the first two. The twinned state is necessary to ensure that the two paths of the bubble share no internal vertex. Whenever $\bar{v}$ is twinned, it can only be explored from v. On the other hand, a blocked vertex should never be explored. A twin vertex $\bar{v}$ can be already blocked when the algorithm is exploring v, since it could have been unsuccessfully explored by some other call. In this case, it is necessary to verify the status of $\bar{v}$, as it is shown in the graph of Fig. 5.3a. Indeed, consider the algorithm starting from s with (s, a) and (a, b) being the first two arcs visited in the lower part. Later, when the calls CYCLE($\bar{c}$) and CYCLE($\bar{b}$) are made, since $\bar{a}$ is twinned, both $\bar{b}$ and $\bar{c}$ remain blocked. When the algorithm backtracks to a and explores (a, c), the call CYCLE(c) is made and $\bar{c}$ is already blocked.

Another important difference with respect to Johnson's algorithm is that there is a specific order in which the out-neighbourhood of a vertex should be explored. In particular, notice that the order in which Algorithm 18 explores the neighbours of a vertex v is: first the vertices in $N^+(v) \setminus \{\bar{v}\}$ and then $\bar{v}$. A variant of the algorithm

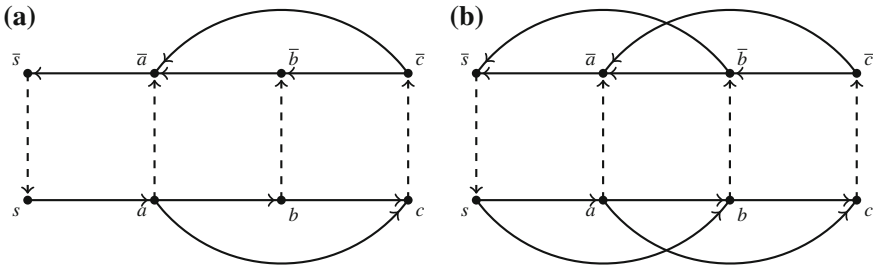


Fig. 5.3 **a** Example where the twin $\bar{v}$ is already blocked when the algorithm starts exploring v. By starting in s and visiting first (s, a) and (a, b), the vertex $\bar{c}$ is already blocked when the algorithm starts exploring c. **b** Counterexample for the variant of the algorithm visiting first the twin and then the regular neighbours. By starting in s and visiting first (s, a) and (a, b), the algorithm misses the bubble-cycle $\langle s, a, c, \bar{c}, \bar{b}, \bar{s}\rangle$

where this order would be reversed, visiting first $\bar{v}$ and then the vertices in $N^+(v)\setminus\{\bar{v}\}$, would fail to enumerate all the bubbles. Indeed, intuitively a vertex can be blocked because the only way to reach $\bar{s}$ is through a twinned vertex and when that vertex is untwinned the first one is not unblocked. Indeed, consider the graph in Fig. 5.3b and the twin-first variant starting in s with (s, a) and (a, b) being the first two arcs explored in the lower part of the graph. When the algorithm starts exploring b the stack contains $\langle s, a, b\rangle$. After, the call CYCLE($\bar{b}$) returns *true* and CYCLE(c) returns *false* because $\bar{a}$ and $\bar{b}$ are twinned. After finishing exploring b, the blocked list $B(b)$ is empty. Thus, the only vertex unblocked is b, c (and $\bar{c}$) remaining blocked. Finally, the algorithm backtracks to a and explores the arc (a, c), but c is blocked, and it fails to enumerate $\langle s, a, c, \bar{c}, \bar{b}, \bar{s}\rangle$.

One way to address the problem above would be to modify the algorithm so that every time a vertex $\bar{v}$ is untwinned, a call to UNBLOCK($\bar{v}$) is made. All the bubble-cycles would be correctly enumerated. However, in this case, it is not hard to find an example where the delay would then no longer be linear. Intuitively, visiting first $N^+(v) \setminus \{\bar{v}\}$ and, then $\bar{v}$, works because every vertex u that was blocked (during the exploration of $N^+(v) \setminus \{\bar{v}\}$) should remain blocked when the algorithm explores $\bar{v}$. Indeed, a bubble would be missed only if there existed a path starting from $\bar{v}$, going to $\bar{s}$ through u and avoiding the twinned vertices. This is not possible if no path from $N^+(v) \setminus \{\bar{v}\}$ to u could be completed into a bubble-cycle by avoiding the twinned vertices, as we will show later on.

5.5 Enumerating Bubbles: An Example

Consider the graph in Fig. 5.2a and its transformation in Fig. 5.2b. We want to enumerate all the bubble-cycles of the graph in Fig. 5.2b by using Algorithm 16 and thus Algorithms 17 and 18. At the beginning every vertex has a status that is free, the

Algorithm 18: Procedure CYCLE(v)

```
   Input: A vertex v ∈ V
   Output: All the bubbles in G starting from v
 1 f ← false;
 2 push v;
 3 status(v) ← blocked;
   /* Exploring forward the arcs going out from v ∈ V            */
 4 if v ∈ V then
 5 |  if status(v̄) = free then status(v̄)← twinned ;
 6 |  for w ∈ N+(v) ∩ V do
 7 |  |  if status(w) = free then
 8 |  |  |  if CYCLE(w) then f ← true;
 9 |  |  end
10 |  end
11 |  if status(v̄) = twinned then
12 |  |  if CYCLE(v̄) then f ← true ;
13 |  end
   /* Exploring forward the arcs going out from v ∈ V̄            */
14 else
15 |  for w ∈ N+(v) do
16 |  |  if w = s̄ then
17 |  |  |  output the cycle composed by the stack followed by s̄ and s;
18 |  |  |  f ← true;
19 |  |  else if status(w) = free then
20 |  |  |  if CYCLE(w) then f ← true;
21 |  |  end
22 |  end
23 end
24 if f then UNBLOCK(v) ;
25 else
26 |  for w ∈ N+(v) do
27 |  |  if v ∉ B(w) then B(w) = B(w) ∪ {v} ;
28 |  end
29 end
30 pop v;
31 return f;
```

stack is empty and Algorithm 18 is called with input s. At this point s is put on the stack, its status is now blocked and the status of $\bar{s}$ is now twinned. Then CYCLE(a), CYCLE(b), CYCLE(c), CYCLE(d) are called in this order, blocking and putting on the stack the vertices a, b, c, d, respectively, and twinning the vertices $\bar{a}$, $\bar{b}$, $\bar{c}$, $\bar{d}$, respectively. Observe that, since s is already blocked, CYCLE(c) does not call again CYCLE(s). At this point the stack is s, a, b, c, d. Since the unique neighbour of d is $\bar{d}$, CYCLE($\bar{d}$) is called and $\bar{d}$ is blocked and put on the stack. This corresponds to the code after line 14 in Algorithm 18. Since $\bar{c}$ is twinned, CYCLE($\bar{c}$) is not called, while, since $\bar{e}$ is free, CYCLE($\bar{e}$) is called and $\bar{e}$ is thus blocked and put on the stack. Once again, since $\bar{b}$ is twinned, CYCLE($\bar{b}$) is not called, and $\bar{s}$ is reached: the cycle containing the vertex on the stack plus $\bar{s}, s, a, b, c, d, \bar{d}, \bar{e}, \bar{s}$, is output.

After the output, the call CYCLE($\bar{e}$) returns and $\bar{e}$ is unblocked and removed from the stack. The same happens for the calls CYCLE($\bar{d}$) and CYCLE(d). At this point the stack is s, a, b, c and CYCLE($\bar{c}$) is called: $\bar{c}$ is thus blocked and put on the stack. Since $\bar{b}$ is twinned, CYCLE($\bar{b}$) is not called, while since $\bar{e}$ is free, CYCLE($\bar{e}$) is called and $\bar{e}$ is thus blocked and put on the stack. Once again, since $\bar{b}$ is twinned, CYCLE($\bar{b}$) is not called, and $\bar{s}$ is reached: the cycle containing the vertex on the stack plus $\bar{s}$, $s, a, b, c, \bar{c}, \bar{e}, \bar{s}$, is output.

After this latter output, the call CYCLE($\bar{e}$), and after CYCLE($\bar{c}$) and CYCLE(c), return, so that $\bar{e}$, $\bar{c}$, and c are unblocked and removed from the stack. Now the stack is s, a, b and we are exploiting other feasible neighbours of b: in particular CYCLE(e) is called. We have that e is now blocked and put on the stack. Since c is free, CYCLE(c) is called; this latter calls CYCLE(d), that calls CYCLE($\bar{d}$). At this point the stack is $s, a, b, e, c, d, \bar{d}$ and it is not possible to complete the stack in order to get a bubble-cycle. Thus f is false, and for any out-neighbour w of $\bar{d}$, v is added to $B(w)$, so that $\bar{d}$ is added to $B(\bar{c})$ and to $B(\bar{e})$. Hence CYCLE($\bar{d}$) returns and, also in CYCLE(d), f remains false, so that $\bar{d}$ is added to $B(d)$. At this point CYCLE(d) returns, we are in CYCLE(c) and we have to finish to exploit the neighbours of c. The stack is s, a, b, e, c, the vertices $s, a, b, c, d, e, \bar{d}$ are blocked, the vertices $\bar{s}, \bar{a}, \bar{b}, \bar{c}, \bar{e}$ are twinned. With this settings, CYCLE(c) calls CYCLE($\bar{c}$) that returns false; $\bar{c}$ is thus added to $B(\bar{b})$ and $B(\bar{e})$. Also CYCLE(c) returns false, so that c is added to $B(d)$ and $B(\bar{c})$. We are thus in CYCLE(e): c has been already considered and d is blocked, CYCLE($\bar{e}$) is thus called. Observe that CYCLE(e) never calls directly CYCLE(d) because it is known that it would return false, thanks to the previous calls from CYCLE(c). This avoids useless computation by realizing the so-called pruning strategy. At this point CYCLE($\bar{e}$) reaches $\bar{s}$ and a bubble-cycle is output: $s, a, b, e, \bar{e}, \bar{s}$. Summarizing, $s, a, b, c, d, e, \bar{c}, \bar{d}$ are blocked, $\bar{s}, \bar{a}, \bar{b}, \bar{e}$ are twinned. In particular $B(d) = \{c\}$, $B(\bar{b}) = \{\bar{c}\}$, $B(\bar{c}) = \{\bar{d}, c\}$, $B(\bar{d}) = \{d\}$, $B(\bar{e}) = \{\bar{d}, \bar{c}\}$, and the other B sets are empty.

After the output, vertex $\bar{e}$ is removed from the stack, unblocked, and any vertex $w \in B(\bar{e})$ is recursively unblocked, so that $\bar{d}$ and $\bar{c}$ are unblocked, and successively also d and c are unblocked. Thus $B(x)$ is empty for any vertex x except for $B(\bar{b}) = \{\bar{c}\}$. CYCLE($\bar{e}$) returns inside CYCLE(e), also e is removed from the stack and unblocked and CYCLE(e) returns inside CYCLE(b). Now the unique blocked vertices are s, a, b and $\bar{s}, \bar{a}, \bar{c}$ are the unique twinned vertices: the algorithm continues by calling CYCLE($\bar{b}$) and after CYCLE($\bar{a}$). Then it returns inside CYCLE($\bar{s}$), where, with empty stack, no blocked vertices, empty B sets, and no twinned vertices, CYCLE(e) is called.

By proceeding in the same way, also the bubble-cycles:

- $s, e, c, \bar{c}, \bar{b}, \bar{a}, \bar{s}$,
- $s, e, d, \bar{d}, \bar{c}, \bar{b}, \bar{a}, \bar{s}$, and
- $s, e, \bar{e}, \bar{b}, \bar{a}, \bar{s}$

are outputted.

5.6 Proof of Correctness and Complexity Analysis

The first part of this section is devoted to prove that Algorithm 18 enumerates all bubbles with source s.

Lemma 5.1 *Let v be a vertex of G'_s such that $status(v) = blocked$, S the set of vertices currently in the stack, and T the set of vertices whose status is equal to twinned. Then $S \cup T$ is a $(v, \overline{s})$ separator, that is, each path, if any exists, from v to $\overline{s}$ contains at least one vertex in $S \cup T$.*

Proof The result is obvious for the vertices in $S \cup T$. Let v be a vertex of G'_s such that $status(v) = blocked$ and $v \notin S \cup T$. This means that when v was popped for the last time, CYCLE(v) was equal to *false* since v remained blocked.

Let us prove by induction on k that each path to $\overline{s}$ of length k from a blocked vertex not in $S \cup T$ contains at least one vertex in $S \cup T$.

We first consider the base case $k = 1$. Suppose that v is a counter-example for $k = 1$. This means that there is an arc from v to $\overline{s}$ ($\overline{s}$ is an out-neighbour of v). However, in that case the output of CYCLE(v) is *true*, a contradiction because v would then be unblocked.

Suppose that the result is true for $k - 1$ and, by contradiction, that there exists a blocked vertex $v \notin S \cup T$ and a path $(v, w, \ldots, \overline{s})$ of length k avoiding $S \cup T$. Since $(w, \ldots, \overline{s})$ is a path of length $k - 1$, we can then assume that w is free. Otherwise, if w were blocked, by induction, the path $(w, \ldots, \overline{s})$ would contain at least one vertex in $S \cup T$, and so would the path $(v, w, \ldots, \overline{s})$.

Since the call to CYCLE(v) returned *false* (v remained blocked), either w was already blocked or twinned, or the call to CYCLE(w) made inside CYCLE(v) gave an output equal to *false*. In any case, after the call to CYCLE(v), w was blocked or twinned and v put in $B(w)$.

The conditional at line 11 of the CYCLE procedure ensures that when untwinned, a vertex immediately becomes blocked. Thus, since w is now free, a call to UNBLOCK(w) was made in any case, yielding a call to UNBLOCK(v). This contradicts the fact that v is blocked.

Theorem 5.1 *The algorithm returns only bubble-cycles. Moreover, each of those cycles is returned exactly once.*

Proof Let us first prove that only bubble-cycles are output. As any call to UNBLOCK (either inside the procedure CYCLE or inside the procedure UNBLOCK itself) is immediately followed by the popping of the considered vertex, no vertex can appear twice in the stack. Thus, the algorithm returns only cycles. They are trivially bipolar as they have to contain s and $\overline{s}$ to be output.

Consider now a cycle C output by the algorithm with swap arc $(t, \overline{t})$. Let (v, w) in C with $v \neq s$ and $v \neq t$. If $\overline{v}$ is free when v is put on the stack, then $\overline{v}$ is twinned before w is put on the stack and cannot be explored until w is popped. If $\overline{v}$ is blocked when v is put on the stack, then by Lemma 5.1 it remains blocked at least until v is popped. Thus, $\overline{v}$ cannot be in C, and consequently the output cycles are twin-free.

So far we have proven that the output produces bubble-cycles. Let us now show that all cycles $C = \{v_0 = s, v_1, \ldots, v_{l-1}, v_l = \overline{s}, v_0\}$ satisfying those conditions are output by the algorithm, and each is output exactly once.

The fact that C is not returned twice is a direct consequence of the fact that the stack is different in all the leaves of a backtracking procedure. To show that C is output, let us prove by induction that the stack is equal to $\{v_0, \ldots, v_i\}$ at some point of the algorithm, for every $0 \leq i \leq l-1$. Indeed, it is true for $i = 0$. Moreover, suppose that at some point, the stack is $\{v_0, \ldots, v_{i-1}\}$.

Suppose that v_{i-1} is different from t. As the cycle contains no pair of twins except for those composing the arcs $(s, \overline{s})$ and $(t, \overline{t})$, the path $\{v_i, v_{i+1}, \ldots, v_l\}$ contains no twin of $\{v_0, \ldots, v_{i-1}\}$ and therefore no twinned vertex. Thus, it is a path from v_i to $\overline{s}$ avoiding $S \cup T$. Lemma 5.1 then ensures that at this point v_i is not blocked. As it is also not twinned, its status is free. Therefore, it will be explored by the backtracking procedure and the stack at some point will be $\{v_0, \ldots, v_i\}$. If $v_{i-1} = t$, $v_i = \overline{t}$ is not blocked using the same arguments. Thus it was twinned by the call to CYCLE(t) and is therefore explored at line 12 of this procedure. Again, the stack at some point will be $\{v_0, \ldots, v_i\}$.

As in [10], we show that Algorithm 18 has delay $O(|V| + |E|)$ by proving that a cycle has to be output between two successive unblockings of the same vertex and that with linear delay some vertex has to be unblocked again. To do so, let us first prove the following lemmas.

Lemma 5.2 *Let v be a vertex such that CYCLE(v) returns true. Then a cycle is output after that call and before any call to UNBLOCK.*

Proof Let y be the first vertex such that UNBLOCK(y) is called inside CYCLE(v). Since CYCLE(v) returns true, there is a call to UNBLOCK(v) before it returns, so that y exists. Certainly, UNBLOCK(y) was called before UNBLOCK(v) if $y \neq v$. Moreover, the call UNBLOCK(y) was done inside CYCLE(y), from line 24, otherwise it would contradict the choice of y. So, the call to CYCLE(y) was done within the recursive calls inside the call to CYCLE(v). CYCLE(y) must then return true as y was unblocked from it.

All the recursive calls CYCLE(z) made inside CYCLE(y) must return false, otherwise there would be a call to UNBLOCK(z) before UNBLOCK(y), contradicting the choice of y. Since CYCLE(y) must return true and the calls to all the neighbours returned false, the only possibility is that $\overline{s} \in N^+(y)$. Therefore, a cycle is output before UNBLOCK(y).

Lemma 5.3 *Let v be a vertex such that there is a $(v, \overline{s})$-path P avoiding $S \cup T$ at the moment a call to CYCLE(v) is made. Then the return value of CYCLE(v) is true.*

Proof First notice that if there is such a path P, then v belongs to a cycle in G'_s. This cycle may however not be a bubble-cycle in the sense that it may not be twin-free, that is, it may contain more than two pairs of twin vertices. Indeed, since the only constraint that we have on P is that it avoids all vertices that are in S and T when v

is reached, then if $v \in V$, it could be that the path P from v to $\overline{s}$ contains, besides s and $\overline{s}$, at least two more pairs of twin vertices. An example is given in Fig. 5.2b. It is however always possible, by construction of G'_s from G, to find a vertex $y \in V$ such that y is the first vertex in P with $\overline{y}$ also in P. Let P' be the path that is a concatenation of the subpath $s \rightsquigarrow y$ of P, the arc $(y, \overline{y})$, and the subpath $\overline{y} \rightsquigarrow \overline{s}$ in P. This path is twin-free, and a call to CYCLE(v) will, by correctness of the algorithm, return true.

Theorem 5.2 *Algorithm 18 has linear delay.*

Proof Let us first prove that between two successive unblockings of any vertex v, a cycle is output. Let w be the vertex such that a call to UNBLOCK(w) at line 24 of Algorithm 18 unblocks v for the first time. Let S and T be, respectively, the current sets of stack and twinned vertices after popping w. The recursive structure of the unblocking procedure then ensures that there exists a (v, w)-path avoiding $S \cup T$. Moreover, as the call to UNBLOCK(w) was made at line 24, the answer to CYCLE(w) is *true* so there exists also a $(w, \overline{s})$-path avoiding $S \cup T$. The concatenation of both paths is a again a $(v, \overline{s})$-path avoiding $S \cup T$. Let x be the first vertex of this path to be visited again. Note that, if no vertex in this path is visited again there is nothing to prove, since v is free, CYCLE(v) needs to be called before any UNBLOCK(v) call. When CYCLE(x) is called, there is a $(x, \overline{s})$-path avoiding the current $S \cup T$ vertices. Thus, applying Lemma 5.3 and then Lemma 5.2, we know that a cycle is output before any call to UNBLOCK. As no call to UNBLOCK(v) can be made before the call to CYCLE(x), a cycle is output before the second call to UNBLOCK(v).

Let us now consider the delay of the algorithm. In both its exploration and unblocking phases, the algorithm follows the arcs of the graph and transforms the status or the B lists of their endpoints, which overall require constant time. Thus, the delay only depends on the number of arcs which are considered during two successive outputs. An arc (u, v) is considered once by the algorithm in the three following situations: the exploration part of a call to CYCLE(u); an insertion of u in $B(v)$; a call to UNBLOCK(v). As shown before, UNBLOCK(v) is called only once between two successive outputs. CYCLE(u) cannot be called more than twice. Thus the arc (u, v) is considered at most 5 times between two outputs. This ensures that the delay of the algorithm is $O(|V| + |E|)$.

5.7 Avoiding Duplicate Bubbles

The one-to-two correspondence between cycles in G'_s and bubbles starting from s in G, claimed by Proposition 5.1, can be reduced to a one-to-one correspondence in the following way. Consider an arbitrary order on the vertices of V, and assign to each vertex of $\overline{V}$ the order of its twin. Let C be a cycle of G'_s that passes through s and contains exactly two pairs of twin vertices. Denote again by t the vertex such that $(t, \overline{t})$ is the arc through which C swaps from V to $\overline{V}$. Denote by *swap predecessor* the vertex before t in C and by *swap successor* the vertex after $\overline{t}$ in C.

Proposition 5.2 *There is a one-to-one correspondence between the set of* (s, t) *-bubbles in* G *for all* $t \in V$*, and the set of cycles of* G'_s *that pass through* s*, contain exactly two pairs of twin vertices and such that the swap predecessor is greater than the swap successor.*

Proof The proof follows the one of Proposition 5.1. The only difference is that, if we consider a bubble composed of the paths P_1 and P_2, one of these two paths, say P_1, has a next to last vertex greater than the next to last vertex of P_2. Then the cycle of G'_s made of P_1 and $\overline{P_2}$ is still considered by the algorithm whereas the cycle made of P_2 and $\overline{P_1}$ is not. Moreover, the cycles of length four which are of the type $\{s, t, \overline{t}, \overline{s}\}$ are ruled out as $\overline{s}$ is of the same order as s.

5.8 Conclusion and Open Problems

We showed in this chapter that it is possible (Algorithm 18) to enumerate all bubbles with a given source in a directed graph with linear delay. Moreover, it is possible to enumerate all bubbles, for all possible sources (Algorithm 16), in $O((|E| + |V|)(|C| + |V|))$ total time, where $|C|$ is the number of bubbles. This required a non trivial adaptation of Johnson's algorithm [10]. The main question arising from our work is whether it is possible to generalize our result, by finding a linear delay algorithm enumerating k-tuple of vertex disjoint paths. Finally we remark that further developments in [99] allow to enumerate bubbles of constrained length in linear delay.

Chapter 6
Enumerating Cycles and (s, t)-Paths in Undirected Graphs

6.1 Introduction

Listing all the simple cycles (hereafter just called cycles) in a graph is a classical problem whose efficient solutions date back to the early 70s. For a graph with n vertices and m edges, containing η cycles, the best known solution in the literature is given by Johnson's algorithm [10] and takes $O((\eta + 1)(m + n))$ time. In the case of biological networks, studying paths or cycles can be useful for several purposes. In the case of interaction graphs, such as gene regulatory networks, the importance of enumeration has been shown in [12], and two algorithms for this problem have been proposed in [53, 12]. As shown in Chap. 3, these networks are directed and their arcs are signed, where the sign or weight of the arcs indicates the causal relationship between the vertices, such as activation or inhibition. In particular, as summarized by [12], cycles and paths can be useful for studying:

- dependencies among vertices: a vertex x activates another vertex y when at least one positive path from x to y exists but no negative one [53].
- the steady state and multistationarity of dynamic models [100, 13, 14].
- monotonicity with respect to changes in the initial conditions [101].

In particular the enumeration of cycles and paths can be useful for investigating:

- feedback loops, that are claimed to be sources of complex dynamics [13, 14]. Moreover feedback loops are related to robustness in cell signalling networks [15].
- signalling paths, by analysing the different positive and negative routes along which a molecule can affect another.
- (Minimal) cut sets: for a given set of feedback loops or signalling paths one may compute a set of interventions interrupting the signal flow in them [53].

In the following we will consider the problem of enumerating paths and cycles in the case of undirected graphs. Our contribution is not just restricted to biological networks, but extends also to arbitrary graph.

A. Marino, *Analysis and Enumeration*, Atlantis Studies in Computing 6,
DOI 10.2991/978-94-6239-097-3_6

Previous Work

The classical problem of listing all the cycles of a graph has been extensively studied for its many applications in several fields, ranging from the mechanical analysis of chemical structures [102] to the design and analysis of reliable communication networks, and the graph isomorphism problem [103]. In particular, at the turn of the seventies several algorithms for enumerating all cycles of an undirected graph have been proposed. There is a vast body of work, and the majority of the algorithms listing all the cycles can be divided into the following three classes (see [104, 105] for excellent surveys).

1. *Search space algorithms*. According to this approach, cycles are looked for in an appropriate search space. In the case of undirected graphs, the *cycle vector space* [106] turned out to be the most promising choice: from a basis for this space, all vectors are computed and it is tested whether they are a cycle. Since the algorithm introduced in [103], many algorithms have been proposed: however, the complexity of these algorithms turns out to be exponential in the dimension of the vector space, and thus in n. For planar graphs, an algorithm listing cycles in $O((\eta + 1)n)$ time was presented in [107].
2. *Backtrack algorithms*. By this approach, all paths are generated by backtrack and, for each path, it is tested whether it is a cycle. One of the first algorithms is the one proposed in [9], which is however exponential in η. By adding a simple pruning strategy, this algorithm has been successively modified in [28]: it lists all the cycles in $O(nm(\eta + 1))$ time. Further improvements were proposed in [10, 108, 29], leading to $O((\eta + 1)(m + n))$-time algorithms that work for both directed and undirected graphs.
3. *Using the powers of the adjacency matrix*. This approach uses the so-called *variable adjacency matrix*, that is, the formal sum of edges joining two vertices. A non-zero element of the pth power of this matrix is the sum of all walks of length p: hence, to compute all cycles, we compute the nth power of the variable adjacency matrix. This approach is not very efficient because of the non-simple walks. Algorithms based on this approach (e.g. [109, 110]) basically differ only on the way they avoid to consider walks that are neither paths nor cycles.

Almost 40 years after Johnson's algorithm [10], the problem of efficiently listing all cycles of a graph is still an active area of research (e.g. [11, 111–116]). Nevertheless, no significant improvement has been obtained from the theory standpoint: in particular, Johnson's algorithm is still the theoretically most efficient. His $O((\eta + 1)(m + n))$-time solution and its linear delay guarantee is surprisingly not optimal for undirected graphs as we show in this chapter.

Contribution

We present the first optimal solution to list all the cycles in an undirected graph G. Specifically, let $\mathscr{C}(G)$ denote the set of all these cycles ($|\mathscr{C}(G)| = \eta$). Our algorithm

requires $O(m+\sum_{c\in\mathscr{C}(G)}|c|)$ time and is asymptotically optimal: indeed, $\Omega(m)$ time is necessarily required to read G as input, and $\Omega(\sum_{c\in\mathscr{C}(G)}|c|)$ time is necessarily required to list the output. Since the length of a cycle $|c|\leq n$, the cost of our algorithm never exceeds $O(m+(\eta+1)n)$ time.

Along the same lines, we also present the first optimal solution to list all the simple paths from s to t (shortly, (s,t)-paths) in an undirected graph G. Let $\mathscr{P}_{st}(G)$ denote the set of (s,t)-paths in G. Our algorithm lists all the (s,t)-paths in G optimally in $O(m+\sum_{\pi\in\mathscr{P}_{st}(G)}|\pi|)$ time, observing that $\Omega(\sum_{\pi\in\mathscr{P}_{st}(G)}|\pi|)$ time is necessarily required to list the output.

We prove the following reduction to relate $\mathscr{C}(G)$ and $\mathscr{P}_{st}(G)$ for some suitable choices of vertices s,t: if there exists an optimal algorithm to list the (s,t)-paths in G, then there exists an optimal algorithm to list the cycles in G. Hence, we can focus on listing (s,t)-paths.

Our work appeared in [16].

Difficult Graphs for Johnson's Algorithm

It is worth observing that the analysis of the time complexity of Johnson's algorithm is not pessimistic and cannot match the one of our algorithm for listing cycles. For example, consider the sparse "diamond" graph $D_n=(V,E)$ in Fig. 6.1 with $n=2k+3$ vertices in $V=\{a,b,c,v_1,\ldots,v_k,u_1,\ldots,u_k\}$. There are $m=\Theta(n)$ edges in $E=\{(a,c),(a,v_i),(v_i,b),(b,u_i),(u_i,c),$ for $1\leq i\leq k\}$, and three kinds of (simple) cycles: (1) $(a,v_i),(v_i,b),(b,u_j),(u_j,c),(c,a)$ for $1\leq i,j\leq k$; (2) $(a,v_i),(v_i,b),(b,v_j),(v_j,a)$ for $1\leq i<j\leq k$; (3) $(b,u_i),(u_i,c),(c,u_j),(u_j,b)$ for $1\leq i<j\leq k$, totalizing $\eta=\Theta(n^2)$ cycles. Our algorithm takes $\Theta(n+k^2)=\Theta(\eta)=\Theta(n^2)$ time to list these cycles. On the other hand, Johnson's algorithm takes $\Theta(n^3)$ time, and the discovery of the $\Theta(n^2)$ cycles in (1) costs $\Theta(k)=\Theta(n)$ time each: the backtracking procedure in Johnson's algorithm starting at a, and passing through v_i, b and u_j for some i,j, arrives at c: at that point, it explores all the vertices u_l $(l\neq i)$ even if they do not lead to cycles when coupled with a, v_i, b, u_j, and c.

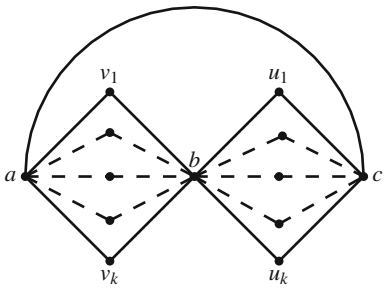


Fig. 6.1 Diamond graph

Structure of the Chapter
This chapter is organised as follows: after introducing the main definitions and notations in Sect. 6.2, in Sect. 6.3 we show the main ideas of our algorithm; in Sect. 6.4 the general amortization strategy of our analysis is reported and in Sect. 6.5 the maintenance of the certificate, a data structure used by the algorithm, is described; in Sect. 6.6 we show how the algorithm works through an example and in Sect. 6.7 we explain in detail the operations performed by the algorithm and its analysis; finally, we conclude in Sect. 6.8.

6.2 Preliminaries

Let $G = (V, E)$ be an undirected connected graph with $n = |V|$ vertices and $m = |E|$ edges, without self-loops or parallel edges. Recall that $\mathscr{P}(G)$ is the set of all paths in G and $\mathscr{P}_{s,t}(G)$ is the set of all (s, t)-paths in G. When $s = t$ we have cycles, and $\mathscr{C}(G)$ denotes the set of all cycles in G. In this chapter, given an undirected graph $G = (V, E)$ we consider the problems of listing all the cycles $c \in \mathscr{C}(G)$ (*listing Cycles*) and all the paths $\pi \in \mathscr{P}_{s,t}(G)$ between two given distinct vertices $s, t \in V$ (*listing* (s, t)*-Paths*).

Our algorithms assume without loss of generality that the input graph G is connected, hence $m \geq n - 1$, and use the decomposition of G into biconnected components. Recall that an *articulation point* (or cut-vertex) is a vertex $u \in V$ such that the number of connected components in G increases when u is removed. G is *biconnected* if it has no articulation points. Otherwise, G can always be decomposed into a tree of biconnected components, called the *block tree*, where each biconnected component is a maximal biconnected subgraph of G (see Fig. 6.2), and two biconnected components are adjacent if and only if they share an articulation point.

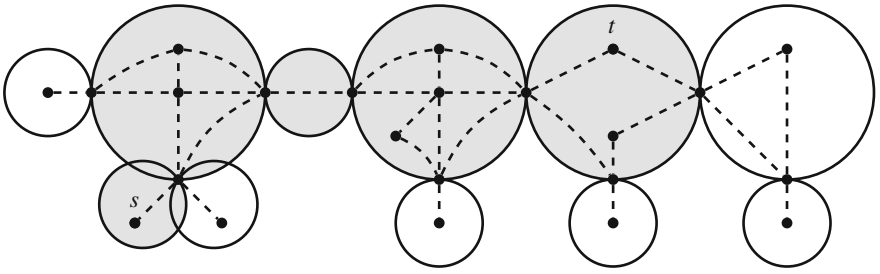


Fig. 6.2 Block tree of G with bead string $B_{s,t}$ in *gray*

6.3 Overview and Main Ideas

While the basic approach is simple (see the binary partition in point 3), we use a number of non-trivial ideas to obtain our optimal algorithm for an undirected (connected) graph G as summarized in the steps below.

1. Prove the following reduction. If there exists an optimal algorithm to list the (s,t)-paths in G, there exists an optimal algorithm to list the cycles in G. This relates $\mathscr{C}(G)$ and $\mathscr{P}_{st}(G)$ for some choices s,t.
2. Focus on listing the (s,t)-paths. Consider the decomposition of the graph into biconnected components (BCCs), thus forming a tree T where two BCCs are adjacent in T iff they share an articulation point. Exploit (and prove) the property that if s and t belong to distinct BCCs, then (i) there is a unique *sequence* $B_{s,t}$ of adjacent BCCs in T through which each (s,t)-path must necessarily pass, and (ii) each (s,t)-path is the concatenation of paths connecting the articulation points of these BCCs in $B_{s,t}$.
3. Recursively list the (s,t)-paths in $B_{s,t}$ using the classical binary partition (i.e. given an edge e in G, list all the cycles containing e, and then all the cycles not containing e): now it suffices to work on the *first* BCC in $B_{s,t}$, and efficiently maintain it when deleting an edge e, as required by the binary partition.
4. Use a notion of *certificate* to avoid recursive calls (in the binary partition) that do not list new (s,t)-paths. This certificate is maintained dynamically as a data structure representing the first BCC in $B_{s,t}$, which guarantees that there exists at least one *new* solution in the current $B_{s,t}$.
5. Consider the binary recursion tree corresponding to the binary partition. Divide this tree into *spines*: a spine corresponds to the recursive calls generated by the edges e belonging to the same adjacency list in $B_{s,t}$. The amortized cost for each listed (s,t)-path π is $O(|\pi|)$ when there is a guarantee that the amortized cost in each spine S is $O(\mu)$, where μ is a lower bound on the number of (s,t)-paths that will be listed from the recursive calls belonging to S. The (unknown) parameter μ, which is different for each spine S, and the corresponding cost $O(\mu)$, will drive the design of the proposed algorithms.

6.3.1 Reduction to Paths

We now show that listing cycles reduces to listing (s,t)-paths while preserving the optimal complexity.

Lemma 6.1 *Given an algorithm that solves the problem of* listing (s, t)-Paths *in optimal* $O(m + \sum_{\pi \in \mathscr{P}_{s,t}(G)} |\pi|)$ *time, there exists an algorithm that solves the problem of* listing Cycles *in optimal* $O(m + \sum_{c \in \mathscr{C}(G)} |c|)$ *time.*

Proof Compute the biconnected components of G and keep them in a list L. Each (simple) cycle is contained in one of the biconnected components and therefore we

can treat each biconnected component individually as follows. While L is not empty, extract a biconnected component $B = (V_B, E_B)$ from L and repeat the following three steps: (i) compute a DFS traversal of B and take any back edge $b = (s, t)$ in B; (ii) list all (s, t)-paths in $B - b$, i.e. the cycles in B that include edge b; (iii) remove edge b from B, compute the new biconnected components thus created by removing edge b, and append them to L. When L becomes empty, all the cycles in G have been listed.

Creating L takes $O(m)$ time. For every $B \in L$, steps (i) and (iii) take $O(|E_B|)$ time. Note that step (ii) always outputs distinct cycles in B (i.e. (s, t)-paths in $B - b$) in $O(|E_B| + \sum_{\pi \in \mathscr{P}_{s,t}(B-b)} |\pi|)$ time. However, $B - b$ is then decomposed into biconnected components whose edges are traversed again. We can pay for the latter cost: for any edge $e \neq b$ in a biconnected component B, there is always a cycle in B that contains both b and e (i.e. it is an (s, t)-path in $B - b$), hence $\sum_{\pi \in \mathscr{P}_{s,t}(B-b)} |\pi|$ dominates the term $|E_B|$, i.e. $\sum_{\pi \in \mathscr{P}_{s,t}(B-b)} |\pi| = \Omega(|E_B|)$. Therefore steps (i)–(iii) take $O(\sum_{\pi \in \mathscr{P}_{s,t}(B-b)} |\pi|)$ time. When L becomes empty, the whole task has taken $O(m + \sum_{c \in \mathscr{C}(G)} |c|)$ time.

6.3.2 Decomposition in Biconnected Components

We now focus on listing (s, t)-paths. We use the decomposition of G into a block tree of biconnected components. Given vertices s, t, define its *bead string*, denoted by $B_{s,t}$, as the unique sequence of one or more adjacent biconnected components (the *beads*) in the block tree, such that the first one contains s and the last one contains t (see Fig. 6.2): these biconnected components are connected through articulation points, which must belong to all the paths to be listed.

Lemma 6.2 *All the (s, t)-paths in $\mathscr{P}_{s,t}(G)$ are contained in the induced subgraph $G[B_{s,t}]$ for the bead string $B_{s,t}$. Moreover, all the articulation points in $G[B_{s,t}]$ are traversed by each of these paths.*

Proof Consider an edge $e = (u, v)$ in G such that $u \in B_{s,t}$ and $v \notin B_{s,t}$. Since the biconnected components of a graph form a tree and the bead string $B_{s,t}$ is a path in this tree, there are no paths $v \rightsquigarrow w$ in $G - e$ for any $w \in B_{s,t}$ because the biconnected components in G are maximal and there would be a larger one (a contradiction). Moreover, let $B_1, B_2, \ldots, B_r$ be the biconnected components composing $B_{s,t}$, where $s \in B_1$ and $t \in B_r$. If there is only one biconnected component in the path (i.e. $r = 1$), there are no articulation points in $B_{s,t}$. Otherwise, all of the $r - 1$ articulation points in $B_{s,t}$ are traversed by each path $\pi \in \mathscr{P}_{s,t}(G)$: indeed, the articulation point between adjacent biconnected components B_i and B_{i+1} is their only vertex in common and there are no edges linking B_i and B_{i+1}.

We thus restrict the problem of listing the paths in $\mathscr{P}_{s,t}(G)$ to the induced subgraph $G[B_{s,t}]$, conceptually isolating it from the rest of G. For the sake of description, we will use interchangeably $B_{s,t}$ and $G[B_{s,t}]$ in the rest of the chapter.

6.3.3 Binary Partition Scheme

We list the set of (s, t)-paths in $B_{s,t}$, denoted by $\mathscr{P}_{s,t}(B_{s,t})$, by applying the binary partition method (where $\mathscr{P}_{s,t}(G) = \mathscr{P}_{s,t}(B_{s,t})$ by Lemma 6.2): we choose an edge $e = (s, v)$ incident to s and then list all the (s, t)-paths that include e and then all the (s, t)-paths that do not include e. Since we delete some vertices and some edges during the recursive calls, we proceed as follows.

Invariant: At a generic recursive step on vertex u (initially, $u := s$), let $\pi_s = s \rightsquigarrow u$ be the path discovered so far (initially, π_s is empty {}). Let $B_{u,t}$ be the current bead string (initially, $B_{u,t} := B_{s,t}$). More precisely, $B_{u,t}$ is defined as follows: (i) remove from $B_{s,t}$ all the vertices in π_s but u, and the edges incident to u and discarded so far; (ii) recompute the block tree on the resulting graph; (iii) $B_{u,t}$ is the unique bead string that connects u to t in the recomputed block tree.

Base case: When $u = t$, output the (s, t)-path π_s.

Recursive rule: Let $\mathscr{P}(\pi_s, u, B_{u,t})$ denote the set of (s, t)-paths to be listed by the current recursive call. Then, it is the union of the following two disjoint sets, for an edge $e = (u, v)$ incident to u:

- *Left branching:* the (s, t)-paths in $\mathscr{P}(\pi_s \cdot e, v, B_{v,t})$ that use e, where $B_{v,t}$ is the unique bead string connecting v to t in the block tree resulting from the deletion of vertex u from $B_{u,t}$.
- *Right branching:* the (s, t)-paths in $\mathscr{P}(\pi_s, u, B'_{u,t})$ that do *not* use e, where $B'_{u,t}$ is the unique bead string connecting u to t in the block tree resulting from the deletion of edge e from $B_{u,t}$.

Hence, $\mathscr{P}_{s,t}(B_{s,t})$ (and so $\mathscr{P}_{s,t}(G)$) can be computed by invoking $\mathscr{P}(\{\}, s, B_{s,t})$. The correctness and completeness of the above approach is discussed in Sect. 6.3.4.

At this point, it should be clear why we introduce the notion of bead strings in the binary partition. The existence of the partial path π_s and the bead string $B_{u,t}$ guarantees that there surely exists at least one (s, t)-path. But there are two sides of the coin when using $B_{u,t}$.

1. One advantage is that we can avoid useless recursive calls: If vertex u has only one incident edge e, we just perform the left branching; otherwise, we can safely perform both the left and right branching since the *first* bead in $B_{u,t}$ is always a biconnected component by definition (thus there exists both an (s, t)-path that traverses e and one that does not).
2. The other side of the coin is that we have to maintain the bead string $B_{u,t}$ as $B_{v,t}$ in the left branching and as $B'_{u,t}$ in the right branching by Lemma 6.2. Note that these bead strings are surely non-empty since $B_{u,t}$ is non-empty by induction (we only perform either left or left/right branching when there are solutions by item 1).

To efficiently address point 2, we need to introduce the notion of certificate as described next.

6.3.4 Introducing the Certificate

Given the bead string $B_{u,t}$, we call the *head* of $B_{u,t}$, denoted by H_u, the first biconnected component in $B_{u,t}$, where $u \in H_u$. Consider a DFS tree of $B_{u,t}$ rooted at u that changes along with $B_{u,t}$, and classify the edges in $B_{u,t}$ as tree edges or back edges (there are no cross edges since the graph is undirected).

To maintain $B_{u,t}$ (and so H_u) during the recursive calls, we introduce a *certificate* C (see Fig. 6.3): It is a suitable data structure that uses the above classification of the edges in $B_{u,t}$, and supports the following operations, required by the binary partition scheme.

Algorithm 19: `list_paths`$_{s,t}(\pi_s,\ u,\ C)$

```
1  if u = t then
2      output(π_s) ;
3      return ;
4  end
5  e = (u, v) ← choose(C, u) ;
6  if e is back edge then
7      I ← right_update(C, e) ;
8      list_paths_{s,t}(π_s, u, C) ;
9      restore(C, I) ;
10 end
11 I ← left_update(C, e) ;
12 list_paths_{s,t}(π_s · (u, v), v, C) ;
13 restore(C, I) ;
```

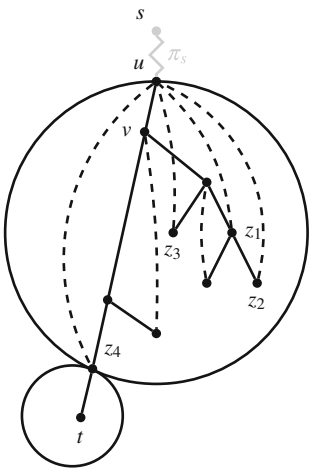


Fig. 6.3 Example certificate of $B_{u,t}$

- `choose`(C, u): returns an edge $e = (u, v)$ with $v \in H_u$ such that $\pi_s \cdot (u, v) \cdot u \rightsquigarrow t$ is an (s, t)-path such that $u \rightsquigarrow t$ is inside $B_{u,t}$. Note that e always exists since H_u is biconnected. Also, the chosen v is the last one in DFS postorder among the neighbours of u: in this way, the (only) tree edge e is returned when there are no back edges leaving from u. (As it will be clear in Sects. 6.4 and 6.5, this order facilitates the analysis and the implementation of the certificate.)
- `left_update`(C, e): for the given $e = (u, v)$, it obtains $B_{v,t}$ from $B_{u,t}$ as discussed in Sect. 6.3.3. This implies updating also H_u, C, and the block tree, since the recursion continues on v. It returns bookkeeping information I for what is updated, so that it is possible to revert to $B_{u,t}$, H_u, C, and the block tree, to their status before this operation.
- `right_update`(C, e): for the given $e = (u, v)$, it obtains $B'_{u,t}$ from $B_{u,t}$ as discussed in Sect. 6.3.3, which implies updating also H_u, C, and the block tree. It returns bookkeeping information I as in the case of `left_update`(C, e).
- `restore`(C, I): reverts the bead string to $B_{u,t}$, the head H_u, the certificate C, and the block tree, to their status before operation $I :=$ `left_update`(C, e) or $I :=$ `right_update`(C, e) was issued (in the same recursive call).

Note that a notion of certificate in listing problems has been introduced in [117], but it cannot be directly applied to our case due to the different nature of the problems and our use of more complex structures such as biconnected components.

Using our certificate and its operations, we can now formalize the binary partition and its recursive calls $\mathscr{P}(\pi_s, u, B_{u,t})$ described in Sect. 6.3.3 as Algorithm 19, where $B_{u,t}$ is replaced by its certificate C.

The base case ($u = t$) corresponds to lines 1–4 of Algorithm 19. During recursion, the left branching corresponds to lines 5 and 11–13, while the right branching to lines 5–10. Note that we perform only the left branching when there is only one incident edge in u, which is a tree edge by definition of `choose`. Also, lines 9 and 13 are needed to restore the parameters to their values when returning from the recursive calls.

Lemma 6.3 *Algorithm* 19 *correctly lists all the* (s, t)*-paths in* $\mathscr{P}_{s,t}(G)$.

Proof For a given vertex u the function `choose`(C, u) returns an edge e incident to u. We maintain the invariant that π_s is a path $s \rightsquigarrow u$, since at the point of the recursive call in line 12: (i) is connected as we append edge (u, v) to π_s and; (ii) it is simple as vertex u is removed from the graph G in the call to `left_update`(C, e) in line 11. In the case of recursive call in line 8 the invariant is trivially maintained as π_s does not change. The algorithm only outputs (s, t)-paths since π_s is a $s \rightsquigarrow u$ path and $u = t$ when the algorithm outputs, in line 2.

The paths with prefix π_s that do not use e are listed by the recursive call in line 8. This is done by removing e from the graph (line 7) and thus no path can include e. Paths that use e are listed in line 12 since in the recursive call e is added to π_s. Given that the tree edge incident to u is the last one to be returned by `choose`(C, u), there is no path that does not use this edge, therefore it is not necessary to call line 8 for this edge.

A natural question is what is the time complexity: we must account for the cost of maintaining C and for the cost of the recursive calls of Algorithm 19. Since we cannot always maintain the certificate in $O(1)$ time, the ideal situation for attaining an optimal cost is taking $O(\mu)$ time if at least μ (s,t)-paths are listed in the current call (and its nested calls). Unfortunately, we cannot estimate μ efficiently and cannot design Algorithm 19 so that it takes $O(\mu)$ adaptively. We circumvent this by using a different cost scheme in Sect. 6.3.5 that is based on the recursion tree induced by Algorithm 19. Section 6.5 is devoted to the efficient implementation of the above certificate operations according to the cost scheme that we discuss next.

6.3.5 Recursion Tree and Cost Amortization

We now show how to distribute the costs among the several recursive calls of Algorithm 19 so that optimality is achieved. Consider a generic execution on the bead string $B_{u,t}$. We trace this execution by using a binary recursion tree R. The nodes of R are labelled by the arguments of Algorithm 19: specifically, we denote a node in R by the triple $x = \langle \pi_s, u, C \rangle$ iff it represents the call with arguments π_s, u, and C.[1] The left branching is represented by the left child, and the right branching (if any) by the right child of the current node.

Lemma 6.4 *The binary recursion tree R for $B_{u,t}$ has the following properties:*

1. *There is a one-to-one correspondence between the paths in $\mathscr{P}_{s,t}(B_{u,t})$ and the leaves in the recursion tree rooted at node $\langle \pi_s, u, C \rangle$.*
2. *Consider any leaf and its corresponding (s,t)-path π: there are $|\pi|$ left branches in the corresponding root-to-leaf trace.*
3. *Consider the instruction $e :=$ `choose`(C,u) in Algorithm 19: unary (i.e. single-child) nodes correspond to left branches (e is a tree edge) while binary nodes correspond to left and right branches (e is a back edge).*
4. *The number of binary nodes is $|\mathscr{P}_{s,t}(B_{u,t})| - 1$.*

Proof We proceed in order as follows.

1. We only output a solution in a leaf and we only do recursive calls that lead us to a solution. Moreover every node partitions the set of solutions in the ones that use an edge and the ones that do not use it. This guarantees that the leaves in the left subtree of the node corresponding to the recursive call and the leaves in the right subtree do not intersect. This implies that different leaves correspond to different paths from s to t, and that for each path there is a corresponding leaf.
2. Each left branch corresponds to the inclusion of an edge in the path π.
3. Since we are in a biconnected component, there is always a left branch. There can be no unary node as a right branch: indeed for any edge of $B_{u,t}$ there exists always a path from s to t passing through that edge. Since the tree edge is always the last

[1] For clarity, we use "nodes" when referring to R and "vertices" when referring to $B_{u,t}$.

one to be chosen, unary nodes cannot correspond to back edges and binary nodes are always back edges.
4. It follows from point 1 and from the fact that the recursion tree is a binary tree. (In any binary tree, the number of binary nodes is equal to the number of leaves minus 1.)

We define a *spine* of R to be a subset of R's nodes linked as follows: the first node is a node x that is either the left child of its parent or the root of R, and the other nodes are those reachable from x by right branching in R. Let $x = \langle \pi_s, u, C \rangle$ be the first node in a spine S. The nodes in S correspond to the edges that are incident to vertex u in $B_{u,t}$: hence their number equals the degree $d(u)$ of u in $B_{u,t}$, and the deepest (last) node in S is always a tree edge in $B_{u,t}$ while the others are back edges. Figure 6.4 shows the spine corresponding to $B_{u,t}$ in Fig. 6.3. Summing up, R can be seen as composed by spines, unary nodes, and leaves where each spine has a unary node as deepest node. This gives a global picture of R that we now exploit for the analysis.

We define the *compact head*, denoted by $H_X = (V_X, E_X)$, as the (multi)graph obtained by compacting the maximal chains of degree-2 vertices, except u, t, and the vertices that are the leaves of its DFS tree rooted at u.

The rationale behind the above definition is that the costs defined in terms of H_X amortize well, as the size of H_X and the number of (s,t)-paths in the subtree of R rooted at node $x = \langle \pi_s, u, C \rangle$ are intimately related (see Lemma 6.6 in Sect. 6.4) while this is not necessarily true for H_u.

Recall that each leaf corresponds to a path π and each spine corresponds to a compact head $H_X = (V_X, E_X)$. We now define the following abstract cost for spines, unary nodes, and leaves of R, for a sufficiently large constant $c_0 > 0$, that Algorithm 19 must fulfill:

$$T(r) = \begin{cases} c_0 & \text{if } r \text{ is unary} \\ c_0|\pi| & \text{if } r \text{ is a leaf} \\ c_0(|V_X| + |E_X|) & \text{if } r \text{ is a spine} \end{cases} \tag{6.1}$$

Lemma 6.5 *The sum of the costs in the nodes of the recursion tree* $\sum_{r \in R} T(r) = O(\sum_{\pi \in \mathscr{P}_{s,t}(B_{u,t})} |\pi|)$.

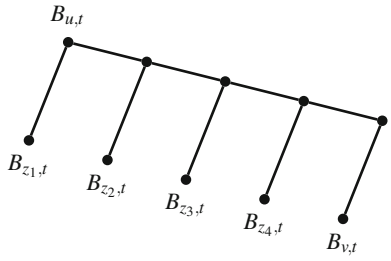


Fig. 6.4 Spine of the recursion tree

Section 6.4 contains the Proof of Lemma 6.5 and related properties. Setting $u := s$, we obtain that the cost in Lemma 6.5 is optimal, by Lemma 6.2.

Theorem 6.1 *Algorithm 19 lists all the (s, t)-paths in optimal $O(m+\sum_{\pi\in\mathscr{P}_{s,t}(G)}|\pi|)$ time.*

By Lemma 6.1, we obtain an optimal result for listing cycles.

Theorem 6.2 *The cycles of an undirected graph can be optimally listed in $O(m+\sum_{c\in\mathscr{C}(G)}|c|)$ time.*

6.4 Amortization Strategy

We devote this section to prove Lemma 6.5. Let us split the sum in Eq. (6.1) in three parts, and bound each part individually, as

$$\sum_{r\in R} T(r) \le \sum_{r:\,\text{unary}} T(r) + \sum_{r:\,\text{leaf}} T(r) + \sum_{r:\,\text{spine}} T(r). \tag{6.2}$$

We have that $\sum_{r:\,\text{unary}} T(r) = O(\sum_{\pi\in\mathscr{P}_{s,t}(G)}|\pi|)$, since there are $|\mathscr{P}_{s,t}(G)|$ leaves, and the root-to-leaf trace leading to the leaf for π contains at most $|\pi|$ unary nodes by Lemma 6.4, where each unary node has cost $O(1)$ by Eq. (6.1).

Also, $\sum_{r:\,\text{leaf}} T(r) = O(\sum_{\pi\in\mathscr{P}_{s,t}(G)}|\pi|)$, since the leaf r for π has cost $O(|\pi|)$ by Eq. (6.1).

It remains to bound $\sum_{r\ \text{spine}} T(r)$. By Eq. (6.1), we can rewrite this cost as $\sum_{H_X} c_0(|V_X|+|E_X|)$, where the sum ranges over the compacted heads H_X associated with the spines r. We use the following lemma to provide a lower bound on the number of (s,t)-paths descending from r.

Lemma 6.6 *Given a spine r, and its bead string $B_{u,t}$ with head H_u, there are at least $|E_X|-|V_X|+1$ (s,t)-paths in G that have prefix $\pi_s = s \rightsquigarrow u$ and suffix $u \rightsquigarrow t$ internal to $B_{u,t}$, where the compacted head is $H_X = (V_X, E_X)$.*

Proof H_X is biconnected. In any biconnected graph $B = (V_B, E_B)$ there are at least $|E_B|-|V_B|+1$ xy-paths for any $x, y \in V_B$. Find an ear decomposition [106] of B and consider the process of forming B by adding ears one at the time, starting from a single cycle including x and y. Initially $|V_B| = |E_B|$ and there are 2 xy-paths. Each new ear forms a path connecting two vertices that are part of a xy-path, increasing the number of paths by at least 1. If the ear has k edges, its addition increases V by $k-1$, E by k, and the number of xy-paths by at least 1. The result follows by induction.

The implication of Lemma 6.6 is that there are at least $|E_X|-|V_X|+1$ leaves descending from the given spine r. Hence, we can charge to each of them a cost of $\frac{c_0(|V_X|+|E_X|)}{|E_X|-|V_X|+1}$. Lemma 6.7 allows us to prove that the latter cost is $O(1)$ when H_u is different from a single edge or a cycle. (If H_u is a single edge or a cycle, H_X is a single or double edge, and the cost is trivially a constant.)

Lemma 6.7 *For a compacted head $H_X = (V_X, E_X)$, its density is $\frac{|E_X|}{|V_X|} \geq \frac{11}{10}$.*

Proof Consider the following partition $V_X = \{r\} \cup V_2 \cup V_3$ where: r is the root; V_2 is the set of vertices with degree 2 and; V_3, the vertices with degree ≥ 3. Since H_X is compacted DFS tree of a biconnected graph, we have that V_2 is a *subset* of the leaves and V_3 contains the set of internal vertices (except r). There are no vertices with degree 1 and $d(r) \geq 2$. Let $x = \sum_{v \in V_3} d(v)$ and $y = \sum_{v \in V_2} d(v)$. We can write the density as a function of x and y, namely,

$$\frac{|E_X|}{|V_X|} = \frac{x + y + d(r)}{2(|V_3| + |V_2| + 1)}$$

Note that $|V_3| \leq \frac{x}{3}$ as the vertices in V_3 have at least degree 3, $|V_2| = \frac{y}{2}$ as vertices in V_2 have degree exactly 2. Since $d(r) \geq 2$, we derive the following bound

$$\frac{|E_X|}{|V_X|} \geq \frac{x + y + 2}{\frac{2}{3}x + y + 2}$$

Consider any graph with $|V_X| > 3$ and its DFS tree rooted at r. Note that: (i) there are no tree edges between any two leaves, (ii) every vertex in V_2 is a leaf and (iii) no leaf is a child of r. Therefore, every tree edge incident in a vertex of V_2 is also incident in a vertex of V_3. Since exactly half the incident edges to V_2 are tree edges (the other half are back edges) we get that $y \leq 2x$.

With $|V_X| \geq 3$ there exists at least one internal vertex in the DFS tree and therefore $x \geq 3$.

$$\begin{aligned} \text{minimize} \quad & \frac{x + y + 2}{\frac{2}{3}x + y + 2} \\ \text{subject to} \quad & 0 \leq y \leq 2x, \\ & x \geq 3. \end{aligned}$$

Since for any x the function is minimized by the maximum y s.t. $y \leq 2x$ and for any y by the minimum x, we get

$$\frac{|E_X|}{|V_X|} \geq \frac{9x + 6}{8x + 6} \geq \frac{11}{10}.$$

Specifically, let $\alpha = \frac{11}{10}$ and write $\alpha = 1 + 2/\beta$ for a constant β: we have that $|E_X| + |V_X| = (|E_X| - |V_X|) + 2|V_X| \leq (|E_X| - |V_X|) + \beta(|E_X| - |V_X|) = \frac{\alpha+1}{\alpha-1}$ $(|E_X| - |V_X|)$. Thus, we can charge each leaf with a cost of $\frac{c_0(|V_X|+|E_X|)}{|E_X|-|V_X|+1} \leq c_0 \frac{\alpha+1}{\alpha-1} =$ $O(1)$. This motivates the definition of H_X, since Lemma 6.7 does not necessarily hold for the head H_u (due to the unary nodes in its DFS tree).

One last step to bound $\sum_{H_X} c_0(|V_X| + |E_X|)$: as noted before, a root-to-leaf trace for the string storing π has $|\pi|$ left branches by Lemma 6.4, and as many spines,

each spine charging $c_0\frac{\alpha+1}{\alpha-1} = O(1)$ to the leaf at hand. This means that each of the $|\mathscr{P}_{s,t}(G)|$ leaves is charged for a cost of $O(|\pi|)$, thus bounding the sum as $\sum_{r \text{ spine}} T(r) = \sum_{H_X} c_0(|V_X| + |E_X|) = O(\sum_{\pi \in \mathscr{P}_{s,t}(G)} |\pi|)$. This completes the Proof of Lemma 6.5. As a corollary, we obtain the following result.

Lemma 6.8 *The recursion tree R with cost as in Eq. (6.1) induces an $O(|\pi|)$ amortized cost for each (s, t)-path π.*

6.5 Certificate Implementation and Maintenance

The certificate C associated with a node $\langle \pi_s, u, C \rangle$ in the recursion tree is a compacted and augmented DFS tree of bead string $B_{u,t}$, rooted at vertex u. The DFS tree changes over time along with $B_{u,t}$, and is maintained in such a way that t is in the leftmost path of the tree. We compact the DFS tree by contracting the vertices that have degree 2, except u, t, and the leaves (the latter surely have incident back edges). Maintaining this compacted representation is not a difficult data-structure problem. From now on we can assume w.l.o.g. that C is an augmented DFS tree rooted at u where internal nodes of the DFS tree have degree ≥ 3, and each vertex v has associated the following information.

1. A doubly-linked list $lb(v)$ of back edges linking v to its descendants w sorted by postorder DFS numbering.
2. A doubly-linked list $ab(v)$ of back edges linking v to its ancestors w sorted by preorder DFS numbering.
3. An integer $\gamma(v)$, such that if v is an ancestor of w then $\gamma(v) < \gamma(w)$.
4. The smallest $\gamma(w)$ over all w, such that (h, w) is a back edge and h is in the subtree of v, denoted by $lowpoint(v)$.

Given three vertices $v, w, x \in C$ such that v is the parent of w and x is not in the subtree[2] of w, we can efficiently test if v is an articulation point, i.e. $lowpoint(w) \leq \gamma(v)$. (Note that we adopt a variant of *lowpoint* using $\gamma(v)$ in place of the preorder numbering [118]: it has the same effect whereas using $\gamma(v)$ is preferable since it is easier to dynamically maintain.)

Lemma 6.9 *The certificate associated with the root of the recursion can be computed in $O(m)$ time.*

Proof In order to set t to be in the leftmost path, we perform a DFS traversal of graph G starting from s and stop when we reach vertex t. We then compute the DFS tree, traversing the path $s \rightsquigarrow t$ first. When visiting vertex v, we set $\gamma(v)$ to depth of v in the DFS. Before going up on the traversal, we compute the lowpoints using the lowpoints of the children. Let z be the parent of v. If $lowpoint(v) \leq \gamma(z)$ and v is not

[2] The second condition is always satisfied when w is not in the leftmost path, since t is not in the subtree of w.

in the leftmost path in the DFS, we cut the subtree of v as it does not belong to $B_{s,t}$. When first exploring the neighbourhood of v, if w was already visited, i.e. $e = (u, w)$ is a back edge, and w is a descendant of v; we add e to $ab(w)$. This maintains the DFS preordering in the ancestor back edge list. Now, after the first scan of $N(v)$ is over and all the recursive calls returned (all the children were explored), we re-scan the neighbourhood of v. If $e = (v, w)$ is a back edge and w is an ancestor of v, we add e to $lb(w)$. This maintains the DFS postorder in the descendant back edge list. This procedure takes at most two DFS traversals in $O(m)$ time. This DFS tree can be compacted in the same time bound.

Lemma 6.10 *Operation* `choose`(C, u) *can be implemented in* $O(1)$ *time.*

Proof If the list $lb(v)$ is empty, return the tree edge $e = (u, v)$ linking u to its only child v (there are no other children). Else, return the last edge in $lb(v)$.

We analyse the cost of updating and restoring the certificate C. We can reuse parts of C, namely, those corresponding to the vertices that are not in the compacted head $H_X = (V_X, E_X)$ as defined in Sect. 6.3.5. We prove that, given a unary node u and its tree edge $e = (u, v)$, the subtree of v in C can be easily made a certificate for the left branch of the recursion.

Lemma 6.11 *On a unary node,* `left_update`(C, e) *takes* $O(1)$ *time.*

Proof Take edge $e = (u, v)$. Remove edge e and set v as the root of the certificate. Since e is the only edge incident in v, the subtree v is still a DFS tree. Cut the list of children of v keeping only the first child. (The other children are no longer in the bead string and become part of I.) There is no need to update $\gamma(v)$.

We now devote the rest of this section to show how to efficiently maintain C on a spine. Consider removing a back edge e from u: the compacted head $H_X = (V_X, E_X)$ of the bead string can be divided into smaller biconnected components. Many of those can be excluded from the certificate (i.e. they are no longer in the new bead string, and so they are bookkept in I) and additionally we have to update the lowpoints that change. We prove that this operation can be performed in $O(|V_X|)$ total time on a spine of the recursion tree.

Lemma 6.12 *The total cost of all the operations* `right_update`(C, e) *in a spine is* $O(|V_X|)$ *time.*

Proof In the right branches along a spine, we remove all back edges in $lb(u)$. This is done by starting from the last edge in $lb(u)$, i.e. proceeding in reverse DFS postorder. For back edge $b_i = (z_i, u)$, we traverse the vertices in the path from z_i towards the root u, as these are the only lowpoints that can change. While moving upwards on the tree, on each vertex w, we update $lowpoint(w)$. This is done by taking the endpoint y of the first edge in $ab(w)$ (the back edge that goes the topmost in the tree) and

choosing the minimum between $\gamma(y)$ and the lowpoint of each child[3] of w. We stop when the updated $lowpoint(w) = \gamma(u)$ since it implies that the lowpoint of the vertex can not be further reduced. Note that we stop before u, except when removing the last back edge in $lb(u)$.

To prune the branches of the DFS tree that are no longer in $B_{u,t}$, consider again each vertex w in the path from z_i towards the root u and its parent y. We check if the updated $lowpoint(w) \leq \gamma(y)$ and w is not in the leftmost path of the DFS. If both conditions are satisfied, we have that $w \notin B_{u,t}$, and therefore we cut the subtree of w and keep it in I to restore later. We use the same halting criterion as in the previous paragraph.

The cost of removing all back edges in the spine is $O(|V_X|)$: there are $O(|V_X|)$ tree edges and, in the paths from z_i to u, we do not traverse the same tree edge twice since the process described stops at the first common ancestor of endpoints of back edges b_i. Additionally, we take $O(1)$ time to cut a subtree of an articulation point in the DFS tree.

To compute `left_update`(C, e) in the binary nodes of a spine, we use the fact that in every left branching from that spine, the graph is the same (in a spine we only remove edges incident to u and on a left branch from the spine we remove the vertex u) and therefore its block tree is also the same. However, the certificates on these nodes are not the same, as they are rooted at different vertices. Using the reverse DFS postorder of the edges, we are able to traverse each edge in H_X only a constant number of times in the spine.

Lemma 6.13 *The total cost of all operations* `left_update`(C, e) *in a spine is amortized* $O(|E_X|)$.

Proof Let t' be the last vertex in the path $u \rightsquigarrow t$ s.t. $t' \in V_X$. Since t' is an articulation point, the subtree of the DFS tree rooted in t' is maintained in the case of removal of vertex u. Therefore the only modifications of the DFS tree occur in the compacted head H_X of $B_{u,t}$. Let us compute the certificate C_i: this is the certificate of the left branch of the ith node of the spine where we augment the path with the back edge $b_i = (z_i, u)$ of $lb(u)$ in the order defined by `choose`(C, u).

For the case of C_1, we remove u and rebuild the certificate starting form z_1 (the last edge in $lb(u)$) using the algorithm from Lemma 6.9 restricted to H_X and using t' as target and $\gamma(t')$ as a baseline to γ (instead of the depth). This takes $O(|E_X|)$ time.

For the general case of C_i with $i > 1$ we also rebuild (part) of the certificate starting from z_i using the procedure from Lemma 6.9 but we use information gathered in C_{i-1} to avoid exploring useless branches of the DFS tree. The key point is that, when we reach the first bead in common to both $B_{z_i,t}$ and $B_{z_{i-1},t}$, we only explore edges internal to this bead. If an edge e leaving the bead leads to t, we can reuse a subtree

[3] If $lowpoint(w)$ does not change we cannot pay to explore its children. For each vertex we dynamically maintain a list $l(w)$ of its children that have lowpoint equal to $\gamma(u)$. Then, we can test in constant time if $l(w) \neq \emptyset$ and y is not the root u. If both conditions are true $lowpoint(w)$ changes, otherwise it remains equal to $\gamma(u)$ and we stop.

of C_{i-1}. If e does not lead to t, then it has already been explored (and cut) in C_{i-1} and there is no need to explore it again since it will be discarded. Given the order we take b_i, each bead is not added more than once, and the total cost over the spine is $O(|E_X|)$.

Nevertheless, the internal edges E'_X of the first bead in common between $B_{z_i,t}$ and $B_{z_{i-1},t}$ can be explored several times during this procedure.[4] We can charge the cost $O(|E'_X|)$ of exploring those edges to another node in the recursion tree, since this common bead is the head of at least one certificate in the recursion subtree of the left child of the ith node of the spine. Specifically, we charge the first node in the *leftmost* path of the ith node of the spine that has exactly the edges E'_X as head of its bead string: (i) if $|E'_X| \leq 1$ it corresponds to a unary node or a leaf in the recursion tree and therefore we can charge it with $O(1)$ cost; (ii) otherwise it corresponds to a first node of a spine and therefore we can also charge it with $O(|E'_X|)$. We use this charging scheme when $i \neq 1$ and the cost is always charged in the leftmost recursion path of ith node of the spine. Consequently, we never charge a node in the recursion tree more than once.

Lemma 6.14 *On each node of the recursion tree,* `restore`(C, I) *takes time proportional to the size of the modifications kept in* I.

Proof We use standard data structures (i.e. linked lists) for the representation of certificate C. Persistent versions of these data structures exist that maintain a stack of modifications applied to them and that can restore its contents to their previous states. Given the modifications in I, these data structures take $O(|I|)$ time to restore the previous version of C.

Let us consider the case of performing `left_update`(C, e). We cut at most $O(|V_X|)$ edges from C. Note that, although we conceptually remove whole branches of the DFS tree, we only remove edges that attach those branches to the DFS tree. The other vertices and edges are left in the certificate but, as they no longer remain attached to $B_{u,t}$, they will never be reached or explored. In the case of `right_update`(C, e), we have a similar situation, with at most $O(|E_X|)$ edges being modified along the spine of the recursion tree.

From Lemmas 6.10 and 6.12–6.14, it follows that on a spine of the recursion tree we have the costs: `choose`(u) on each node which is bounded by $O(|V_X|)$ time as there are at most $|V_X|$ back edges in u; `right_update`(C, e), `restore`(C, I) take $O(|V_X|)$ time; `left_update`(C, e) and `restore`(C, I) are charged $O(|V_X| + |E_X|)$ time. We thus have the following result, completing the Proof of Theorem 6.1.

Lemma 6.15 *Algorithm 19 can be implemented with a cost fulfilling Eq.* (6.1)*, thus it takes total* $O(m + \sum_{r \in R} T(r)) = O(m + \sum_{\pi \in \mathscr{P}_{s,t}(B_{u,t})} |\pi|)$ *time.*

[4] Consider the case where $z_i, \ldots, z_j$ are all in the same bead after the removal of u. The bead strings are the same, but the roots $z_i, \ldots, z_j$ are different, so we have to compute the corresponding DFS of the first component $|j - i|$ times.

6.6 Enumerating Paths: An Example

Let us consider the graph in Fig. 6.5. In this case Lemma 6.2 is witnessed by vertex h: indeed the bead string $B_{s,t}$ is B_1, B_2, where B_1 is the graph induced by $\{s, a, b, c, d, e, f, g, h, i\}$ and B_2 is the graph induced by $\{h, l, m, t\}$, and h is an articulation point; all the paths between s and t pass through h.

By considering as V_X and E_X respectively the vertices and edges in B_1, Lemma 6.6 with $u = s$ states that there are at least $|E_X| - |V_X| + 1 = 16 - 10 + 1 = 7$ paths between s and t. Indeed observe that in the DFS in Fig. 6.6, B_1 is already compacted since the unique vertex with degree two is c, that is a leaf. Moreover, observe that $|E_X|/|V_X| = 16/10 \geq 11/10$, as stated by Lemma 6.7.

Lemma 6.9 builds the certificate shown in Fig. 6.6. Observe that for any vertex v, whose children are $z_1, \ldots, z_k$,

$$lowpoint(v) = \min\{lowpoint(z_1), \ldots, lowpoint(z_k), \min_{(v,w)\in ab(v)}\{\gamma(w)\}\}.$$

Moreover the elements in any set lb are put in the ascending order defined by the post-order DFS numbering of their extremes. The elements in any set ab are put in the ascending order defined by the pre-order DFS numbering of their extremes.

The first bipartition is applied by considering the edge (s, g), that is the last edge in the list $lb(s)$. The right branch operation simply deletes this edge, and modifies the certificate by applying the method described in Sect. 6.7.1. In general by deleting an edge, some vertices could not reach t, but this is not the case. Moreover this edge has to be removed also from the ab lists and this can be easily done when removing the edge from lb; finally the *lowpoint*s have to be updated. The unique vertices, whose *lowpoint*s change, are on the path of tree edges from g to the root s. Lemma 6.17, in the following section, states that the sum of these costs along the entire spine of s is $O(|V_X|)$, where V_X is the set of vertices in the bead of s and g (Fig. 6.7).

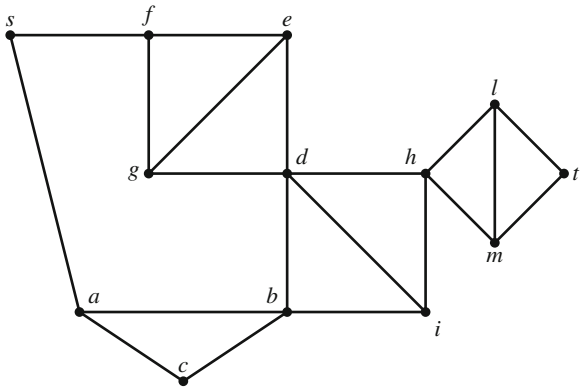


Fig. 6.5 An undirected graph, whose biconnected components are B_1, that is the graph induced by $\{s, a, b, c, d, e, f, g, h, i\}$, and B_2, that is the graph induced by $\{h, l, m, t\}$

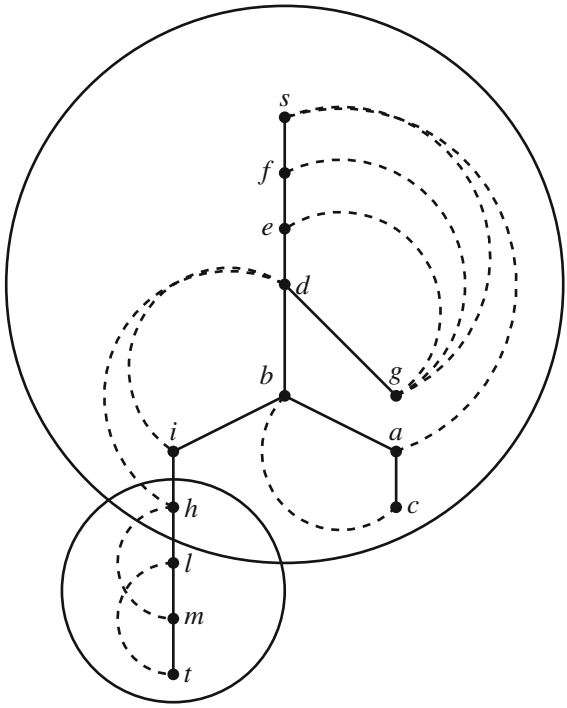


Fig. 6.6 DFS tree starting from s, of the graph in Fig. 6.5, with t in the leftmost path. Each biconnected component is enclosed in a *circle*. For any vertex the properties included by the certificate are reported in the table

Vertex	γ	Post-order Number	Pre-order Number	lb	ab	*lowpoint*
s	0	1	13	(s,a),(s,g)	$\emptyset$	$\gamma(s) = 0$
a	5	11	7	$\emptyset$	(a,s)	$\gamma(s) = 0$
b	4	5	8	(b,c)	$\emptyset$	$\gamma(s) = 0$
c	2	12	6	$\emptyset$	(c,b)	$\gamma(b) = 4$
d	3	4	10	(d,h),(d,i)	$\emptyset$	$\gamma(s) = 0$
e	2	3	11	(e,g)	$\emptyset$	$\gamma(s) = 0$
f	1	2	12	(f,g)	$\emptyset$	$\gamma(s) = 0$
g	4	13	9	$\emptyset$	(g,s),(g,f),(g,e)	$\gamma(s) = 0$
h	6	7	4	(h,m)	(h,d)	$\gamma(d) = 3$
i	5	8	5	$\emptyset$	(i,d)	$\gamma(d) = 3$
l	7	8	3	(l,t)	$\emptyset$	$\gamma(h) = 6$
m	8	9	2	$\emptyset$	(m,h)	$\gamma(h) = 6$
t	9	10	1	$\emptyset$	(t,l)	$\gamma(l) = 7$

The left branch operation removes the vertex s and performs a DFS from g, as shown by Fig. 6.8a. Observe that, since the entry point of the component h, l, m, and t remains h, it is sufficient to perform the visit just inside the biconnected component of s and g. In particular, since this is the first back edge of a spine, i.e. we are computing C_1 referring to Lemma 6.13, we have to rebuild the certificate from

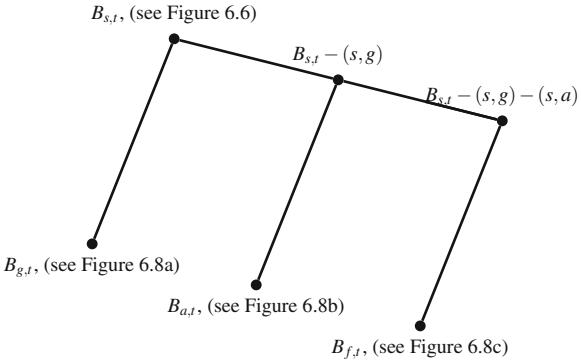


Fig. 6.7 Spine of the recursion tree starting from s

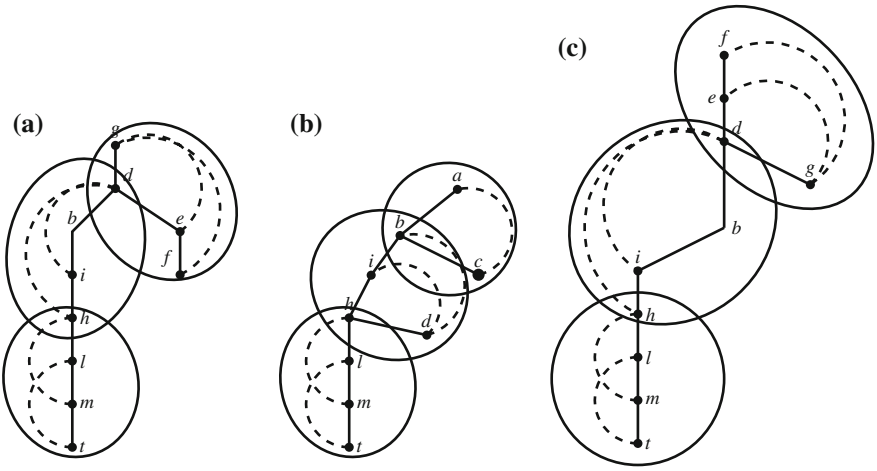


Fig. 6.8 Certificates in the spine of s. **a** $B_{g,t}$. **b** $B_{a,t}$ **c** $B_{f,t}$

scratch by applying Lemma 6.9 (see Lemma 6.20). Observe that the component a, b, c is cut since it does not lead to t and b is compacted.

The second bipartition along the same spine considers the paths from s to t in the graph induced by $B_{s,t} - (s, g)$ and the edge (s, a), that is the remaining edge in $lb(s)$. Once again, the right update operation simply deletes the edge (s, a), removes (s, a) from both lb and ab, updates the *lowpoint*s along the path of tree edges from s to a, and checks whether branches have to be compressed or cut. In this case, by removing the edge (s, a), the vertices a, b, c do not belong anymore to the same biconnected component of s and each path from s to t in $B_{s,t} - (s, g) - (s, a)$, using a or c, should pass more than once through b. This implies that a and c have to be cut (see also Lemma 6.18). By considering the vertices w along the path of tree edges from a to s, that are a, b, d, e, f, s, a vertex y along this path is an articulation point if the updated child w is such that $lowpoint(w) \leq \gamma(y)$ (and there exists at least another

vertex not descendant of y). Once again Lemma 6.18 states that the sum of these costs along the entire spine of s is $O(|V_X|)$, where V_X is the set of vertices in the bead of s and a. Moreover, the vertex b has to be compressed, since by cutting its right subtree, its degree is just two. This operation takes constant time.

By following Lemma 6.13, the left update operation builds the certificate C_2 shown in Fig. 6.8b, by reusing information from C_1. Indeed observe that the certificates C_1, in Fig. 6.8a (plus the cut component a, b, c), and C_2, in Fig. 6.8b, are DFSs defined on the same graph but with different roots. The first bead in common between these two certificates is d, b, i, h. In order to build the certificate in Fig. 6.8b, the information about the components after d, b, i, h in the path of beads from s to t, can be inherited by C_1. However, as stated by Lemma 6.13, the visit of the bead intersection has to be performed, since the entry points, i.e. respectively d and b, are different: this implies that the visit of the component d, b, i, h has to be performed more than once along the spine. Lemma 6.13 (and 6.21) says that these extra costs can be amortized in the cost of the spines of d and b, since, when performing the recursion in d and b, the cost along their spine is at least the size of this component.

The third and last bipartition of the spine considers the tree edge (s, f), by looking for paths from s to t in $B_{s,t} - (s, g) - (s, a)$. Observe that, by deleting (s, g) and (s, a), all the paths from s to t have to pass through f. This implies that the set of paths in $B_{s,t} - (s, g) - (s, a)$ not using the edge (s, f) is empty and the right update operation does not have to be performed. The left update operation shown in Lemma 6.11 takes constant time, since the certificate shown in Fig. 6.8c is simply obtained by $B_{s,t} - (s, g) - (s, a)$ by removing the vertex s and the edge (s, f). If f has two or more children, we should consider just the first child, since it is the unique one leading to t, and cut all the other children; but since f has just one child this operation is not performed. The lb, ab, γ and *lowpoint* do not need to be updated.

6.7 Extended Analysis of Operations

In this section, we present all details and illustrate with figures the operations `right_update`(C, e) and `left_update`(C, e) that are performed along a spine of the recursion tree. In order to better detail the procedures in Lemmas 6.12 and 6.13, we divide them in smaller parts. We use bead string $B_{u,t}$ from Fig. 6.3 and the respective spine from Fig. 6.4 as the base for the examples. This spine contains four binary nodes corresponding to the back edges in $lb(u)$ and an unary node corresponding to the tree edge (u, v). Note that edges are taken in order of the endpoints z_1, z_2, z_3, z_4, v as defined in operation `choose`(C, u).

By Lemma 6.2, the impact of operations `right_update`(C, e) and `left_update`(C, e) in the certificate is restricted to the biconnected component of u. Thus we mainly focus on maintaining the compacted head $H_X = (V_X, E_X)$ of the bead string $B_{u,t}$.

6.7.1 Operation Right_Update(C, e)

Lemma 6.16 (Lemma 6.12 restated) *In a spine of the recursion tree, operations* `right_update`(C, e) *can be implemented in* $O(|V_X|)$ *total time.*

In the right branches along a spine, we remove all back edges in $lb(u)$. This is done by starting from the last edge in $lb(u)$, i.e. proceeding in reverse DFS postorder. In the example from Fig. 6.3, we remove the back edges $(z_1, u) \ldots (z_4, u)$. To update the certificate corresponding to $B_{u,t}$, we have to (i) update the lowpoints in each vertex of H_X; (ii) prune vertices that cease to be in $B_{u,t}$ after removing a back edge. For a vertex w in the tree, there is no need to update $\gamma(w)$.

Consider the update of lowpoints in the DFS tree. For a back edge $b_i = (z_i, u)$, we traverse the vertices in the path from z_i towards the root u. By definition of lowpoint, these are the only lowpoints that can change. Suppose that we remove back edge (z_4, u) in the example from Fig. 6.3, only the lowpoints of the vertices in the path from z_4 towards the root u change. Furthermore, consider a vertex w in the tree that is an ancestor of at least two endpoints z_i, z_j of back edges b_i, b_j. The lowpoint of w does not change when we remove b_i. These observations lead us to the following lemma.

Lemma 6.17 *In a spine of the recursion tree, the update of lowpoints in the certificate by operation* `right_update`(C, e) *can be done in* $O(|V_X|)$ *total time.*

Proof Take each back edge $b_i = (z_i, u)$ in the order defined by `choose`(C, u). Remove b_i from $lb(u)$ and $ab(z_i)$. Starting from z_i, consider each vertex w in the path from z_i towards the root u. On vertex w, we update $lowpoint(w)$ using the standard procedure: take the endpoint y of the first edge in $ab(w)$ (the back edge that goes the nearest to the root of the tree) and choosing the minimum between $\gamma(y)$ and the lowpoint of each child of w. When the updated $lowpoint(w) = \gamma(u)$, we stop examining the path from z_i to u since it implies that the lowpoint of the vertex can not be further reduced (i.e. w is both an ancestor to both z_i and z_{i+1}).

If $lowpoint(w)$ does not change we cannot pay to explore its children. In order to get around this, for each vertex we dynamically maintain, throughout the spine, a list $l(w)$ of its children that have lowpoint equal to $\gamma(u)$. Then, we can test in constant time if $l(w) \neq \emptyset$ and y (the endpoint of the first edge in $ab(w)$) is not the root u. If both conditions are satisfied $lowpoint(w)$ changes, otherwise it remains equal to $\gamma(u)$ and we stop. The total time to create the lists is $O(|V_X|)$ and the time to update is bounded by the number of tree edges traversed, shown to be $O(|V_X|)$ in the next paragraph.

The cost of updating the lowpoints when removing all back edges b_i is $O(|V_X|)$: there are $O(|V_X|)$ tree edges and we do not traverse the same tree edge twice since the process described stops at the first common ancestor of endpoints of back edges b_i and b_{i+1}. By contradiction: if a tree edge (x, y) would be traversed twice when removing back edges b_i and b_{i+1}, it would imply that both x and y are ancestors of z_i and z_{i+1} (as edge (x, y) is both in the path z_i to u and the path z_{i+1} to u) but we stop at the first ancestor of z_i and z_{i+1}.

Let us now consider the removal of vertices that are no longer in $B_{u,t}$ as consequence of operation `right_update`(C, e) in a spine of the recursion tree. By removing a back edge $b_i = (z_i, u)$, it is possible that a vertex w previously in H_X is no longer in the bead string $B_{u,t}$ (e.g. w is no longer biconnected to u and thus there is no simple path $u \rightsquigarrow w \rightsquigarrow t$).

Lemma 6.18 *In a spine of the recursion tree, the branches of the DFS that are no longer in $B_{u,t}$ due to operation* `right_update`(C, e) *can be removed from the certificate in $O(|V_X|)$ total time.*

Proof To prune the branches of the DFS tree that are no longer in H_X, consider again each vertex w in the path from z_i towards the root u and the vertex y, parent of w. It is easy to check if y is an articulation point by verifying if the updated $lowpoint(w) \leq \gamma(y)$ and there exists x not in the subtree of w. If w is not in the leftmost path, then t is not in the subtree of w. If that is the case, we have that $w \notin B_{u,t}$, and therefore we cut the subtree of w and bookkeep it in I to restore later. Like in the update the lowpoints, we stop examining the path z_i towards u in a vertex w when $lowpoint(w) = \gamma(u)$ (the lowpoints and biconnected components in the path from w to u do not change). When cutting the subtree of w, note that there are no back edges connecting it to $B_{u,t}$ (w is an articulation point) and therefore there are no updates to the lists lb and ab of the vertices in $B_{u,t}$. Like in the case of updating the lowpoints, we do not traverse the same tree edge twice (we use the same halting criterion).

With Lemmas 6.17 and 6.18 we finalize the Proof of Lemma 6.12. Figure 6.9 shows the changes the bead string $B_{u,t}$ from Fig. 6.3 goes through in the corresponding spine of the recursion tree.

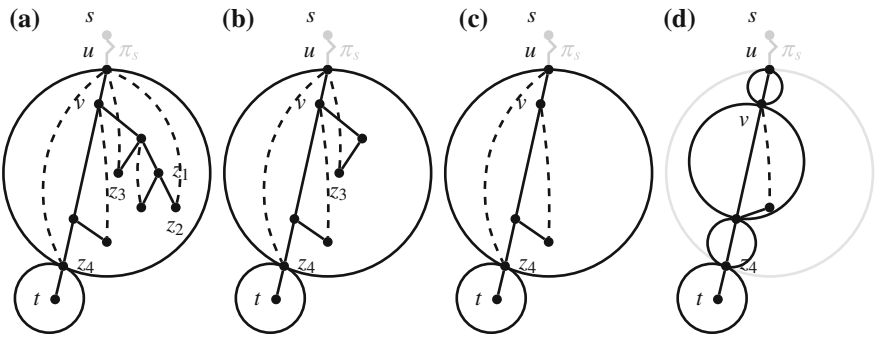


Fig. 6.9 Example application of `right_update`(C, e) on a spine of the recursion tree. **a** Step 1. **b** Step 2. **c** Step 3. **d** Step 4 (final)

6.7.2 Operation Left_Update(C, e)

In the binary nodes of a spine, we use the fact that in every left branching from that spine the graph is the same (in a spine we only remove edges incident to u and on a left branch from the spine we remove the vertex u) and therefore its block tree is also the same. In Fig. 6.10, we show the resulting block tree of the graph from Fig. 6.3 after having removed vertex u. However, the certificates on these left branches are not the same, as they are rooted at different vertices. In the example we must compute the certificates $C_1 \ldots C_4$ corresponding to bead strings $B_{z_1,t} \ldots B_{z_4,t}$. We do not account for the cost of the left branch on the last node of spine (corresponding to $B_{v,t}$) as the node is unary and we have shown in Lemma 6.11 how to maintain the certificate in $O(1)$ time.

By using the reverse DFS postorder of the back edges, we are able to traverse each edge in H_X only an amortized constant number of times in the spine.

Lemma 6.19 (Lemma 6.13 restated) *The calls to operation* `left_update`(C, e) *in a spine of the recursion tree can be charged with a time cost of* $O(|E_X|)$ *to that spine.*

To achieve this time cost, for each back edge $b_i = (z_i, u)$, we compute the certificate corresponding to $B_{z_i,t}$ based on the certificate of $B_{z_{i-1},t}$. Consider the compacted head $H_X = (V_X, E_X)$ of the bead string $B_{u,t}$. We use $O(|E_X|)$ time to compute the first certificate C_1 corresponding to bead string $B_{z_1,t}$. Figure 6.11 shows bead string $B_{z_1,t}$ from the example of Fig. 6.3.

Lemma 6.20 *The certificate* C_1*, corresponding to bead string* $B_{z_1,t}$*, can be computed in* $O(|E_X|)$ *time.*

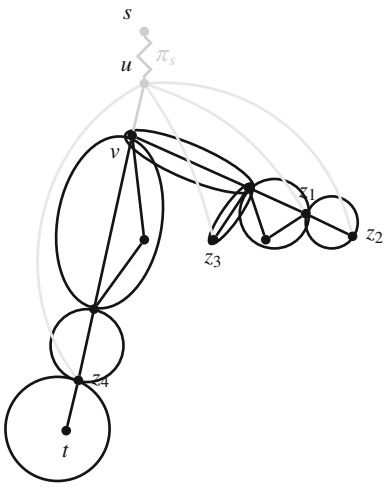


Fig. 6.10 Block tree after removing vertex u

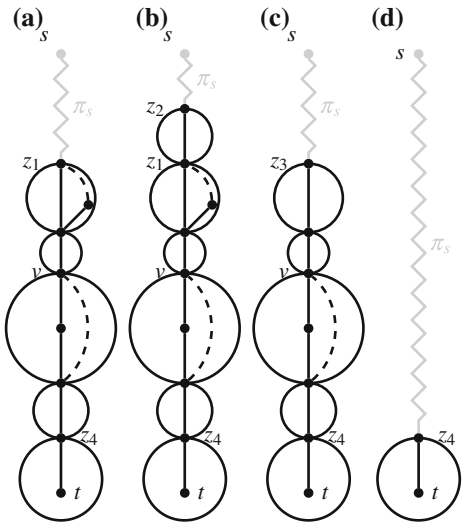


Fig. 6.11 Certificates of the left branches of a spine. **a** $B_{z_1,t}$. **b** $B_{z_2,t}$. **c** $B_{z_3,t}$. **d** $B_{z_4,t}$

Proof Let t' be the last vertex in the path $u \rightsquigarrow t$ s.t. $t' \in V_X$. Since t' is an articulation point, the subtree of the DFS tree rooted in t' is maintained in the case of removal vertex u. Therefore the only modifications of the DFS tree occur in head H_X of $B_{u,t}$.

To compute C_1, we remove u and rebuild the certificate starting form z_1 using the algorithm from Lemma 6.9 restricted to H_X and using t' as target and $\gamma(t')$ as a baseline to γ (instead of the depth). In particular we do the following. To set t' to be in the leftmost path, we perform a DFS traversal of graph H_X starting from z_1 and stop when we reach vertex t'. Then compute the DFS tree, traversing the path $z_1 \rightsquigarrow t'$ first.

Update of γ. For each tree edge (v, w) in the $t' \rightsquigarrow z_1$ path, we set $\gamma(v) = \gamma(w) - 1$, using $\gamma(t')$ as a baseline. During the rest of the traversal, when visiting vertex v, let w be the parent of v in the DFS tree. We set $\gamma(v) = \gamma(w) + 1$. This maintains the property that $\gamma(v) > \gamma(w)$ for any w ancestor of v.

Lowpoints and pruning the tree. Bottom-up in the DFS-tree, compute the lowpoints using the lowpoints of the children. For z the parent of v, if $lowpoint(v) \leq \gamma(z)$ and v is not in the leftmost path in the DFS, cut the subtree of v as it does not belong to $B_{z_1,t}$.

Computing lb and ab. In the traversal, when finding a back edge $e = (v, w)$, if w is a descendant of v we append e to $ab(w)$. This maintains the DFS preorder in the ancestor back edge list. After the first scan of $N(v)$ is over and all the recursive calls returned, re-scan the neighbourhood of v. If $e = (v, w)$ is a back edge and w is an ancestor of v, we add e to $lb(w)$. This maintains the DFS postorder in the descendant back edge list. This procedure takes $O(|E_X|)$ time.

To compute each certificate C_i, corresponding to bead string $B_{z_i,t}$, we are able to avoid visiting most of the edges that belong $B_{z_{i-1},t}$. Since we take z_i in reverse DFS

postorder, on the spine of the recursion we visit $O(|E_X|)$ edges plus a term that can be amortized.

Lemma 6.21 *For each back edge $b_i = (z_i, u)$ with $i > 1$, let E_{X_i}' be the edges in the first bead in common between $B_{z_i,t}$ and $B_{z_{i-1},t}$. The total cost of computing all certificates $B_{z_i,t}$ in a spine of the recursion tree is: $O(|E_X| + \sum_{i>1} |E_{X_i}'|)$.*

Proof Let us compute the certificate C_i: the certificate of the left branch of the ith node of the spine where we augment the path with back edge $b_i = (z_i, u)$ of $lb(u)$.

For the general case of C_i with $i > 1$ we also rebuild (part) of the certificate starting from z_i using the procedure from Lemma 6.9 but we use information gathered in C_{i-1} to avoid exploring useless branches of the DFS tree. The key point is that, when we reach the first bead in common to both $B_{z_i,t}$ and $B_{z_{i-1},t}$, we only explore edges internal to this bead. If an edge e that leaves the bead leads to t, we can reuse a subtree of C_{i-1}. If e does not lead to t, then it has already been explored (and cut) in C_{i-1} and there is no need to explore it again since it is going to be discarded.

In detail, we start computing a DFS from z_i in $B_{u,t}$ until we reach a vertex $t' \in B_{z_{i-1},t}$. Note that the bead of t' has one entry point and one exit point in C_{i-1}. After reaching t' we proceed with the traversal using only edges already in C_{i-1}. When arriving at a vertex w that is not in the same bead of t', we stop the traversal. If w is in a bead towards t, we reuse the subtree of w and use $\gamma(w)$ as a baseline of the numbering γ. Otherwise w is in a bead towards z_{i-1} and we cut this branch of the certificate. When all edges in the bead of t' are traversed, we proceed with visit in the standard way.

Given the order we take b_i, each bead is not added more than once to a certificate C_i, therefore the total cost over the spine is $O(|E_X|)$. Nevertheless, the internal edges E_{X_i}' of the first bead in common between $B_{z_i,t}$ and $B_{z_{i-1},t}$ are explored for each back edge b_i.

Although the edges in E_{X_i}' are in a common bead between $B_{z_i,t}$ and $B_{z_{i-1},t}$, these edges must be visited. The entry point in the common bead can be different for z_i and z_{i-1}, the DFS tree of that bead can also be different. For an example, consider the case where $z_i, \ldots, z_j$ are all in the same bead after the removal of u. The bead strings $B_{z_i,t} \ldots B_{z_j,t}$ are the same, but the roots $z_i, \ldots, z_j$ of the certificate are different, so we have to compute the corresponding DFS of the first bead $|j - i|$ times. Note that this is not the case for the other beads in common: the entry point is always the same.

Lemma 6.22 *The cost $O(|E_X| + \sum_{i>1} |E_{X_i}'|)$ on a spine of the recursion tree can be amortized to $O(|E_X|)$.*

Proof We can charge the cost $O(|E_{X_i}'|)$ of exploring the edges in the first bead in common between $B_{z_i,t}$ and $B_{z_{i-1},t}$ to another node in the recursion tree. Since this common bead is the head of at least one certificate in the recursion subtree of the left child of the ith node of the spine. Specifically, we charge the first and only node in the *leftmost* path of the ith child of the spine that has exactly the edges E_{X_i}' as head of its bead string: (i) if $|E_{X_i}'| \leq 1$ it corresponds to a unary node or a leaf in the recursion

tree and therefore we can charge it with $O(1)$ cost; (ii) otherwise it corresponds to a first node of a spine and therefore we can also charge it with $O(|E'_{X_i}|)$. We use this charging scheme when $i \neq 1$ and the cost is always charged in the leftmost recursion path of ith node of the spine, consequently we never charge a node in the recursion tree more than once.

Lemmas 6.21 and 6.22 finalize the Proof of Lemma 6.13. Figure 6.11 shows the certificates of bead strings $B_{z_i,t}$ on the left branches of the spine from Fig. 6.4.

6.8 Conclusion and Open Problems

We showed in this chapter the first optimal solution to list all the cycles of an undirected graph and all the paths from a given source to a given target. This result improves the Johnson's algorithm, that was still the theoretically most efficient in the case of undirected graphs. The main question arising from our work is whether it is possible to obtain an optimal algorithm to list all the paths and cycles in a directed graph.

Part III
Further Analysis

Chapter 7
Enumerating Diametral and Radial Vertices and Computing Diameter and Radius of a Graph

7.1 Introduction

Structural analysis allows the identification of important and not important vertices within a network and for this reason it has become very popular in many disciplines. In general, the importance of a vertex can be defined in many different ways. The effectiveness of each centrality measure depends on the context of application.

In this chapter, we will focus on the enumeration of the radial and diametral vertices, i.e. central and peripheral vertices according to the eccentricity notion of centrality, and on the computation of the radius and diameter of biological networks and of real world graphs in general. Recall that the diameter D is the maximum distance $d(x, y)$ among all the pairs of vertices x, y. In other words, the diameter of a graph is the maximum forward or backward eccentricity of its vertices, where the forward and backward eccentricities of a vertex x are respectively $ecc_F(x) = max_{y \in V} d(x, y)$ and $ecc_B(x) = max_{y \in V} d(y, x)$. The radius R is instead defined as the minimum forward eccentricity of its vertices. Diametral sources and targets are thus defined as all the vertices x such that $ecc_F(x) = D$ and $ecc_B(x) = D$ respectively; the radial vertices are instead all the vertices x such that $ecc_F(x) = D$.

In the case of an undirected graph, whenever the graph is not connected, i.e. when at least for a pair of vertices x, y there is no path from x to y, the radius and diameter of its connected components are usually studied. In the more general case of a directed graph, whenever the graph is not strongly connected, i.e. at least for a pair of vertices x, y there is no path from x to y, we will consider in this work the radius and diameter of its strongly connected components.

In the case of biological networks, for instance in metabolic networks, the diameter indicates how many reactions have to be performed in order to produce any metabolite from any other metabolite [119]. For several biological networks, it has been studied in [120, 121] and for protein-protein interactions in [122]. The radius can indicate how many reactions have to be performed at least in order to produce all the metabolites from any other metabolite.

A. Marino, *Analysis and Enumeration*, Atlantis Studies in Computing 6,
DOI 10.2991/978-94-6239-097-3_7

The analysis of real world networks in general, such as citations, collaboration, communication, road, social, and web networks, has attracted a lot of attention, and in [17] the fundamental analysis measures have been reviewed. Moreover, the size of these networks has been increasing rapidly, so that, in order to study such measures, algorithms able to handle huge amount of data are needed. Hence the contribution of our algorithms is not just limited to biological networks analysis, but extends to complex networks analysis in general. Indeed, in the case of the diameter, for social networks, in which every vertex is an individual and the edges represent their friendship, the diameter has been studied for several social networks in [120, 123, 124], for peer to peer social networks in [125], for mobile social networks in [126], for Facebook in [22, 23]. For several scientific collaboration networks, in which every vertex is a scientist and scientists are linked whenever they have collaborated for a research paper, the diameter has been studied in [127, 120]. In the case of a web network, in which every vertex corresponds to a web page and the arcs correspond to hyper links, the diameter indicates how quickly any page can be reached. For several web networks, this has been estimated in [120, 128, 129]. Because of the huge size of the networks, in almost any of those works, the diameter of the connected components of undirected graphs or the strongly connected components of directed graphs was just estimated. For these reasons, we have shown the effectiveness of our algorithms not only for biological networks but also for several other kinds of complex networks.

Previous Work

The *Single-source Shortest Path* (in short SSSP) is the problem of finding all the shortest paths from a given vertex to all the others. In general this problem has complexity $O(m)$ in the case of unweighted graphs by using the traditional BFS algorithm and $O((m+n)\log n)$ in the case of weighted graphs, by using the Dijkstra's Algorithm.

In general, algorithms for finding the exact radius or diameter solve the *All Pairs Shortest Path* problem (in short APSP) that is the problem of finding the shortest path between all pairs of vertices of the graph, so that the maximum distance obtained is the diameter. This can be efficiently done by applying the classical *text-book* algorithm, i.e. solving for any vertex the SSSP problem, or by applying fast matrix multiplication with complexity $O(n^{2.376})$ [123]. See [130] for a survey. However, in the context of huge real world networks, these approaches are not practical and usually just estimations or bounds can be provided.

Some algorithms are able to estimate the cumulative distribution of the shortest path lengths of any kind of graph and can be applied to obtain an estimation of the radius and diameter with a small additive error using much less computations with respect to the APSP. This is the case of ANF (Approximate Neighbourhood Function) in [131], HyperANF in [132], HADI in [133], and Cohen frameworks [134–138].

A lower bound of the diameter or an upper bound of the radius can be provided by using a *sample* of the vertices and returning respectively the maximum and the minimum eccentricity found, as done in [139] for the diameter, or by using other heuristics.

In the case of the radius, this sampling method has been exploited for a huge web graph in [128], but as far as we know no other methods have been proposed.

In the case of the diameter, for undirected graphs a lower bound can be provided by using the so called *double sweep* algorithm: pick the farthest vertex from a random vertex and return its eccentricity. The idea can be iterated picking at each step the farthest vertex from the previous one and maintaining the highest eccentricity found as in [120]. In real world networks actually this lower bound is very good and, in order to prove the effectiveness of this approach, several works, like [19, 140], propose strategies to find a matching or close upper bound. Recent advances have shown that in real cases a matching between a lower and upper bound for the diameter can be found by applying a very small number of computations of SSSP, even if, in the worst case, the time complexity degenerates in the time complexity of the APSP problem. In [20], and independently in [141], a lower and upper bound on the diameter are indeed dynamically refined by calculating the eccentricity of vertices properly chosen, so that also the diameter of huge real world undirected graphs has been discovered.

The algorithm shown in [20] has been integrated into the library in [142], and has been used in order to compute the exact diameter of several quite huge subgraphs of the Facebook graph: a highly parallel version of this method was able to compute the diameter of the largest subgraph (approximately 149.1 M of vertices and 15.9 G of edges) in twenty minutes [22, 23].

For directed graphs, in order to obtain a lower bound for the diameter, the idea of the *double sweep* has been adapted by [128]: pick the farthest vertex from a random vertex and return its backward eccentricity, i.e. its eccentricity in the transposed graph. In [18] the effectiveness of this directed version of the *double sweep* has been verified and the technique in [20] has been reviewed and generalized in order to calculate the diameter of directed graphs too.

Contribution

As a result of our previous works [18–21], we will present the D*i*FUB algorithm, which is able to calculate the diameter and to list all the vertices that are sources or targets of a diametral path of (strongly) connected components of huge real world (directed) graphs in time $O(m)$ in practice. Our experimental results are supported by considering an extensive dataset of real world graphs. By using the same technique, we will show an algorithm for efficiently computing the radius and listing all the vertices that are sources of a radial path of such graphs. For the same dataset we will show the effectiveness of this algorithm. It is worth of noting that our algorithms extend also to weighted graphs.

Parts of this chapter appeared in [2, 143].

Structure of the Chapter

This chapter is organised as follows: in Sect. 7.2 we will overview the most popular centrality measures that have been applied to biological networks to discover vertex essentiality; in Sect. 7.3, respectively Sect. 7.4, we will show our algorithm to compute the diameter, respectively the radius, and to list all the peripheral, respectively the

central, vertices according to the eccentricity notion of centrality; in Sect. 7.5 we will show how these algorithms work through an example and in Sect. 7.6, we will report some graphs in which our algorithms achieve the worst performances; in Sect. 7.7 we will show that this is not the case of real world graphs. Finally, in Sect. 7.8 we conclude with some open problems.

7.2 Overview on Centrality Analysis for Biological Networks

Given a network, it is natural to wonder how important each vertex is to the functionality of the network. A number of graph measures have been developed for evaluating vertex centrality [144–149] and several tools allow to compute network metrics, such as CentiBiN [144], VisANT [150], Visone [151], Pajek [152], CentiScaPe [149], and CentiLib [153]. Centrality measures can be local (or neighbourhood based) or global (distance or feedback based).

Local Measures
With neighbourhood-based measures, such as degree, the importance of the vertices is inferred from their local connectivity and the more connections a vertex has the more central it is. Highly connected vertices (hubs) were found to possess special properties in the *Yeast* Protein-Protein Interaction network: they are more often essential than non-hub proteins [154, 155]). They tend to play a central role in the modular organization of a network [156, 65] and seem to be evolutionarily more conserved [157]. Nevertheless, since then, several works have raised doubts on some of these associations [158, 159].

There is no consensus in the literature on how to define a hub, and different criteria have been used: a certain fraction of the highest degree vertices [160]; vertices with a certain fraction of the total connectivity [161]; a degree greater than an arbitrary threshold [65, 162, 163].

In order to have an indication about the homogeneity of the vertices of a network, it is interesting to study the degree distribution that for most biological networks is well fitted by a power-law ($P(k) \propto k^{-\gamma}$) with $\gamma \approx 2$, where k corresponds to the degree. In these networks, a few hubs play a fundamental role for the integrity and navigability of the network [156], while a vast majority of the vertices has only a few connections. This degree distribution has been associated to robustness against random vertex removal. Robustness to the loss of a vertex in the Metabolic network indicates the presence of alternative pathways bypassing the missing reaction; in Gene Regulatory networks, it may correspond to the presence of alternative ways of transducing and controlling information. On the contrary, these networks are highly sensitive to directed attacks, because removal of hubs deeply affects network functionality [164]. Even though much research has been done on the power-law distribution and its universality in biological networks, criticisms have been raised [165].

The local connectivity of vertices can be studied in further detail by using either assortativity or dyadicity. The first measure is the correlation between the degree of adjacent vertices [166]. Maslov and Sneppen [167] found that hubs in the *Yeast* Protein-Protein Interaction network are mostly connected to non-hubs, and are therefore well separated from each other. Dyadicity [168] measures the degree to which vertices of a network are connected to vertices sharing some characteristic (functional classification, essentiality, involvement in a disease and so on) and is therefore able to characterize the modular structure of a network, considering the distribution of the functions over the vertices and their connectivity [169]. A network is called heterophilic (heterophobic) when different categories are connected more (less) often than expected following a random model. It has been recently used to study the coupling between structure and functionality in transcriptional and non coding (nc) RNA-protein interactions networks [170]. The results showed that most transcriptional regulators and ncRNAs tend to connect to genes/proteins of other functional classes, suggesting that regulators do not really belong to a functional class and tend to coordinate several of them [170]. On the converse, in Protein-Protein Interaction networks connections more often involve proteins of a same functional category.

Global Measures

Closeness [171], eccentricity, and shortest path based betweenness [172] are based on global properties of a network, in particular to the shortest path length between its vertices. The closeness of a vertex depends on its average distance from the others and is of particular interest for information networks (such as signalling and gene regulatory networks), because it measures how fast information flows from a vertex of interest to all the reachable vertices on the average [173]. It has been recently integrated with biological information in a parameter-free gene prioritization approach that measures the interconnectedness (ICN) between genes in a network [174]. ICN measures closeness of each candidate gene to genes possessing the interesting property by considering alternative paths in addition to the direct link and the shortest path distances. The closeness can be efficiently approximated in the case of big networks by using [175].

The eccentricity of a vertex is the length of the longest shortest path starting from it, so that a vertex is central if its farthest vertex is not far. It has been shown that in the metabolic network of *E. coli* the rank order of the vertices based on eccentricity yields very similar rank order to the one based on the closeness close to the central vertices, despite the fact that these measures may disagree significantly in the case of not central vertices [176]; for the protein network of *Yeast*, it has been shown that even if eccentricity is not effective to find essential proteins because not essential proteins can have high eccentricity, the proteins having high eccentricity can be considered not essential [176, 177]. In the following section we will exploit an algorithm to find efficiently all the peripheral and central vertices, according to this notion of centrality.

Shortest path based betweenness depends on the number of shortest paths crossing a vertex. In Protein-Protein Interaction networks, betweenness can be interpreted as the relevance of a protein to be intermediary in the interaction between other

proteins, by assuming that this interaction passes through shortest paths [149]. Bottlenecks are vertices with high betweenness centrality and have been found to be key connectors with surprising functional and dynamical properties, often essential [178]. Bottleneck and hub genes were identified in coexpression networks inferred from experimental data, and found to be often essential for virulence in *Salmonella typhimurium* with the role of mediators of transitions between different cellular states or of sentinels that reflect the dynamics of these transitions [179]. Cell cycle checkpoints were found to be bottlenecks in a gene coexpression network of cell cycle regulated genes in the fission *Yeast* [180].

Network metrics in general [181–183] and betweenness centrality in particular are also used for the rational prediction of drug targets [184]. Essential genes are preferred targets for drug design and central genes are more likely to be essential. Another constraint was imposed in this particular case: the gene must be essential for the pathogen but not for the host, to reduce side effects of the drug.

One problem of shortest path based measures is that communication between biological entities is assumed to pass along those paths, which is often not plausible: from the point of view of Metabolic networks, the shortest path might be defined on the basis of the energy/cofactor requirements instead of the number of hops, while in Gene Regulatory networks and Protein-Protein Interaction networks all active connections will take place and not only the shortest ones. In the former, targets with different shortest paths to a common regulator may exhibit hierarchical gene expression patterns as it is the case for flagellar genes [185].

To overcome the limitation of shortest paths, a vertex can be considered central when it is crossed by many random walks: this is the case of the random walk based betweenness centrality [186]. Some feedback based measures are implicitly based on random walks, like eigenvector [187] and spectral centrality [188]. Eigenvector centrality has been applied to several metabolic networks [189] and has been shown to outperform other metrics for the identification of essential proteins in the Protein-Protein Interaction network of *Yeast* [190], together with subgraph centrality [191].

However, since the network express just the potential links and not the real ones, many walks are not feasible, since they traverse edges that are hardly occur together at the same time in the network. For these reasons, very recently, gene expression has been integrated in a centrality measure called Pec [192] which has been used to identify essential genes in *Yeast*. This measure exploits the strength of the connectivity between two adjacent vertices based on an Edge Clustering Coefficient [193], weighted by the co-expression between genes in experimental data.

7.3 Computing the Diameter and Enumerating All the Diametral Vertices

Let $G = (V, E)$ be a directed strongly connected graph and let u be any vertex in V. Let $F_i^F(u)$ be the *forward fringe* of u, that is, the set of vertices x such that $d(u, x) = i$. Similarly, let $F_i^B(u)$ be the *backward fringe*, that is, the set of vertices

x such that $d(x, u) = i$. In other words, $F_i^F(u)$ (respectively, $F_i^B(u)$) includes all vertices at level i of T_u^F (respectively, T_u^B), where recall that T_u^F (respectively, T_u^B) is the forward (respectively, backward) BFS tree.

Remark 7.1 For any two integers i, j with $1 \leq i \leq ecc_B(u)$ and $1 \leq j \leq ecc_F(u)$, for any two vertices x, y such that $x \in F_i^B(u)$ and $y \in F_j^F(u)$, $d(x, y) \leq i + j \leq 2\max\{i, j\}$.

Indeed, since $x \in F_i^B(u)$ and $y \in F_j^F(u)$, there exists a path from x to y passing through u that is long $i + j$, so that $i + j$ is an upper bound for $d(x, y)$.

Theorem 7.1 *For any integer i with $1 < i \leq \mathrm{ecc}_B(u)$, for any integer k with $1 \leq k < i$, and for any vertex $x \in F_{i-k}^B(u)$ such that $\mathrm{ecc}_F(x) > 2(i-1)$, there exists $y \in F_j^F(u)$, for some $j \geq i$, such that $d(x, y) = \mathrm{ecc}_F(x)$. Moreover for any y such that $d(x, y) = \mathrm{ecc}_F(x)$, $y \in F_j^F(u)$ for some $j \geq i$.*

Proof Since $\mathrm{ecc}_F(x) > 2(i-1)$, there exists y such that $d(x, y) > 2(i-1)$. For any y such that $d(x, y) > 2(i-1)$, if y was in $F_j^F(u)$ with $j < i$, then from Remark 7.1 it would follow that $d(x, y) \leq 2\max\{i-k, j\} \leq 2\max\{i-k, i-1\} = 2(i-1)$, which is a contradiction. Hence, y must be in $F_j^F(u)$ with $j \geq i$.

Similarly to the proof of Theorem 7.1, we can also prove the following symmetrical result.

Theorem 7.2 *For any integer i with $1 < i \leq \mathrm{ecc}_F(u)$, for any integer k with $1 \leq k < i$, and for any vertex $x \in F_{i-k}^F(u)$ such that $\mathrm{ecc}_B(x) > 2(i-1)$, there exists $y \in F_j^B(u)$, for some $j \geq i$, such that $d(y, x) = \mathrm{ecc}_B(x)$. Moreover for any y such that $d(y, x) = \mathrm{ecc}_B(x)$, $y \in F_j^B(u)$, for some $j \geq i$.*

In order to describe the DiFUB algorithm, we also need the following definitions. Let

$$B_j^F(u) = \begin{cases} \max_{x \in F_j^F(u)} \mathrm{ecc}_B(x) & \text{if } j \leq \mathrm{ecc}_F(u), \\ 0 & \text{otherwise} \end{cases}$$

and

$$B_j^B(u) = \begin{cases} \max_{x \in F_j^B(u)} \mathrm{ecc}_F(x) & \text{if } j \leq \mathrm{ecc}_B(u), \\ 0 & \text{otherwise.} \end{cases}$$

By using these two definitions, we are now ready to introduce the DiFUB algorithm, which is shown in Algorithm 20. Intuitively, Theorems 7.1 and 7.2 suggest to perform a forward and a backward BFS from a vertex u, and to visit T_u^F and T_u^B in a bottom-up fashion, starting from the vertices in the last fringes. For each level i, we compute the eccentricities of all the vertices in the corresponding fringes: if the maximum eccentricity found lb is greater than $2(i-1)$ then we can conclude that lb is the diameter, since the eccentricities of all the vertices of the remaining levels cannot be greater than lb.

Algorithm 20: DiFUB to compute the diameter

Input: A strongly connected di-graph G, a vertex u, a lower bound l for the diameter
Output: The diameter D

```
1  i ← max{ecc_F(u), ecc_B(u)};
2  lb ← max{ecc_F(u), ecc_B(u), l};
3  ub ← 2i;
4  while ub − lb > 0 do
5  |  lb ← max{lb, B_i^B(u), B_i^F(u)};
6  |  if lb > 2(i − 1) then
7  |  |  return lb;
8  |  end
9  |  ub ← 2(i − 1);
10 |  i ← i − 1;
11 end
12 return lb;
```

Theorem 7.3 *Algorithm 20 correctly computes the value of the diameter of G.*

Proof Let D' be the value returned by Algorithm 20. Note that the diameter cannot be smaller than D' since this value is the length of a shortest path. By contradiction assume that there exists a vertex x such that $ecc_F(x) > D'$. Let j be the last value of i for which Algorithm 21 has computed $B_i^B(u)$ and $B_i^F(u)$: thus $D' \geq 2(j-1)$. For any vertex v in $F_j^B(u) \cup F_{j+1}^B(u) \cup \cdots \cup F_{\mathrm{ecc}_B(u)}^B(u)$, we have computed $ecc_F(v)$, and for any vertex w in $F_j^F(u) \cup F_{j+1}^F(u) \cup \cdots \cup F_{\mathrm{ecc}_F(u)}^F(u)$, we have computed $ecc_B(w)$, lb is the maximum eccentricity found. This implies that x have to belong to $F_h^B(u)$ for some $h < j$. Since $ecc_F(x) > D' \geq 2(j-1)$, from Theorem 7.1, there exists $y \in F_k^F(u)$, for some $k \geq j$, such that $d(x, y) = ecc_F(x)$. In other words there exists $y \in F_j^F(u) \cup F_{j+1}^F(u) \cup \cdots \cup F_{\mathrm{ecc}_F(u)}^F(u)$, such that $ecc_B(y) \geq ecc_F(x) > D'$. Since D' is the maximum eccentricity found, this is a contradiction.

In order to present the algorithm to enumerate all the diametral sources and targets, we define the following quantities, $S_j^F(u)$, that is the set of vertices belonging to $F_j^F(u)$ whose backward eccentricity is $B_j^F(u)$, and the set of their farthest vertices in the transposed graph $T_j^F(u)$.

$$S_j^F(u) = \begin{cases} \{x \in F_j^F(u) : \mathrm{ecc}_B(x) = B_j^F(u)\} & \text{if } j \leq \mathrm{ecc}_F(u), \\ \emptyset & \text{otherwise} \end{cases}$$

$$T_j^F(u) = \bigcup_{x \in S_j^F(u)} \{y : d(y, x) = B_j^F(u)\}$$

Analogously, we define $S_j^B(u)$, that is the set of vertices belonging to $F_j^B(u)$ whose forward eccentricity is $B_j^B(u)$, and the set of their farthest vertices $T_j^B(u)$.

$$S_j^B(u) = \begin{cases} \{x \in F_j^B(u) : \mathrm{ecc}_F(x) = B_j^B(u)\} & \text{if } j \leq \mathrm{ecc}_B(u), \\ \emptyset & \text{otherwise.} \end{cases}$$

$$T_j^B(u) = \bigcup_{x \in S_j^B(u)} \{y : d(x, y) = B_j^B(u)\}$$

With respect to Algorithm 20, every time the lower bound lb is updated, because vertices, whose forward or backward eccentricity is greater than lb are found in the current fringe sets, Algorithm 21 empties and updates also the set of diametral sources DS and the set of diametral targets DT. Every time vertices, whose forward or backward eccentricity is equal to the current value of lb, these vertices are added to the sets DS and DT, respectively. It is worth observing that whenever the guarding condition of the loop is satisfied, because $lb = ub = 2i$, for some i, the eccentricities of the vertices belonging to $F_i^F(u)$ or $F_i^B(u)$ have not been checked, while some of these vertices could have still forward or backward eccentricity equal to $2i$. Thus this check is done in the last part of the algorithm before the returning statement.

Theorem 7.4 *Algorithm 21 correctly computes all the diametral sources and targets of G.*

Proof Algorithm 21 performs at least all the visits performed by Algorithm 20 and return the maximum eccentricity found. From Theorem 7.3, it follows that Algorithm 21 returns the diameter D. The following invariant holds: for any vertex $v \in DS$, $ecc_F(v) \geq lb$; indeed $v \in S_i^B(u)$ and $ecc_F(v) = lb$ or $v \in T_i^F(u)$ for some i and $ecc_F(v) \geq lb$. Since lb is finally the diameter, all the vertices in DS are diametral sources. Let us prove that all the diametral sources are in DS. By contradiction, assume that there exists a vertex x such that $x \notin DS$ and $\mathrm{ecc}_F(x) = D$. Let j be the last value of i for which Algorithm 21 has computed $B_i^B(u)$ and $B_i^F(u)$. Thus for any vertex v in $F_j^B(u) \cup F_{j+1}^B(u) \cup \cdots \cup F_{\mathrm{ecc}_B(u)}^B(u)$, we have computed $ecc_F(v)$, and for any vertex w in $F_j^F(u) \cup F_{j+1}^F(u) \cup \cdots \cup F_{\mathrm{ecc}_F(u)}^F(u)$, we have computed $ecc_B(w)$. Observe that for any pair of vertices v, w such that $d(v, w) = D$, Algorithm 21 is such that, if v belongs to $F_h^B(u)$ for some $h \geq j$, v is added to DS and w is added to DT, if w belongs to $F_k^F(u)$ for some $k \geq j$, w is added to DT and v is added to DS. Thus x have to belong to $F_h^B(u)$ for some $h < j$ and any y, such that $d(x, y) = D$, have to belong to $F_k^F(u)$ for some $k < j$. By applying Theorem 7.1, if $D = ecc_F(x) > 2(j-1)$ then y should belong to $F_z^F(u)$ for some $z \geq j$: thus $D \leq 2(j-1)$. There are the following two cases.

- Algorithm 21 stops and returns inside the loop. In this case, $B_j^F(u)$ and $B_j^B(u)$ are the last computed and $lb = D > 2(j-1)$, that is a contradiction.
- Algorithm 21 stops and returns outside the loop. In this case $B_j^F(u)$ and $B_j^B(u)$ are the last computed and $lb = ub = D = 2j$, that is a contradiction.

Algorithm 21: DiFUB to enumerate all the diametral vertices

Input: A strongly connected di-graph G, a vertex u, a lower bound l for the diameter
Output: The diameter D, DS, that is the set of vertices x such that $\text{ecc}_F(x) = D$, DT, that is the set of vertices x such that $\text{ecc}_B(x) = D$.

```
1  i ← max{ecc_F(u), ecc_B(u)};
2  lb ← max{ecc_F(u), ecc_B(u), l};
3  ub ← 2i;
4  DS ← ∅;  DT ← ∅;
5  while ub − lb > 0 do
6      if B_i^B(u) > lb then
7          lb ← B_i^B(u);
8          DS ← S_i^B(u);  DT ← T_i^B(u);
9      else
10         if B_i^B(u) = lb then
11             DS ← DS ∪ S_i^B(u);  DT ← DT ∪ T_i^B(u);
12         end
13     end
14     if B_i^F(u) > lb then
15         lb ← B_i^F(u);
16         DS ← T_i^F(u);  DT ← S_i^F(u);
17     else
18         if B_i^F(u) = lb then
19             DS ← DS ∪ T_i^F(u);  DT ← DT ∪ S_i^F(u);
20         end
21     end
22     if lb > 2(i − 1) then
23         return lb, DS, DT;
24     end
25     ub ← 2(i − 1);
26     i ← i − 1;
27 end
28 if B_i^B(u) = lb then
29     DS ← DS ∪ S_i^B(u);  DT ← DT ∪ T_i^B(u);
30 end
31 if B_i^F(u) = lb then
32     DS ← DS ∪ T_i^F(u);  DT ← DT ∪ S_i^F(u);
33 end
34 return lb, DS, DT;
```

Analogously, it is possible to prove that DT contains all and only the diametral targets.

Observe that in order to compute $B_j^F(u)$ (respectively, $B_j^B(u)$), we need to compute $ecc_B(x)$ (respectively, $ecc_F(x)$) for any node x in $F_j^F(u)$ (respectively, $F_j^B(u)$), by performing a visit.

The time complexity of DiFUB can be in the worst case $O(nm)$ where n denotes the number of vertices and m denotes the number of arcs. Indeed, observe that,

at each iteration of the **while** loop, $ub - lb$ decreases at least by 2: this implies that, given a starting vertex u, the algorithm executes at most $\max\{\lceil \mathrm{ecc}_B(u)/2 \rceil, \lceil \mathrm{ecc}_F(u)/2 \rceil\}$ iterations (note that we have that the number of iterations is bounded by $D/2$); in the worst case, the number of nodes in $F_j^F(u)$ for $j > \lceil \mathrm{ecc}_F(u)/2 \rceil$ or in $F_i^B(u)$ for $i > \lceil \mathrm{ecc}_B(u)/2 \rceil$ is linear and for each of these nodes a visit is required (see Sect. 7.6). In the case of Algorithm 21, one iteration more could be needed.

Since the practical performance of the algorithm depends on the chosen vertex u, the idea behind a good choice of the starting vertex is preferring vertices having a small quantity of vertices at distance greater than or equal to $D/2$. We will consider the following heuristics to choose the starting vertex and to get a corresponding lower bound l.

- *Degree selection.* A simple way of selecting u is choosing a vertex with the highest in-degree or out-degree. We refer to the composition of DiFUB with these two selection strategies as DiFUB*HdOut* and DiFUB*HdIn* respectively.
- *2-Sweep selection.* A more complex way to select u is by using the following heuristic, called 2*d*SWEEP, which is a natural extension to directed graphs of the 2SWEEP method (in the following, the *middle* vertex between two vertices s and t is defined as the vertex belonging to the shortest path from s to t, whose distance from s is $\lceil d(s,t)/2 \rceil$).

  1. Run a forward BFS from a vertex r: let a_1 be the farthest vertex.
  2. Run a backward BFS from a_1: let b_1 be the farthest vertex.
  3. Run a backward BFS from r: let a_2 be the farthest vertex.
  4. Run a forward BFS from a_2: let b_2 be the farthest vertex.
  5. If $\mathrm{ecc}_B(a_1) > \mathrm{ecc}_F(a_2)$, then set u equal to the middle vertex between a_1 and b_1 and l equal to $\mathrm{ecc}_B(a_1)$. Otherwise, set u equal to the middle vertex between a_2 and b_2 and l equal to $\mathrm{ecc}_F(a_2)$.

  We will consider the variants in which r is the vertex with highest out-degree or in-degree, and we will refer to them as 2*d*SWEEP*HdOut* and 2*d*SWEEP*HdIn* respectively. Moreover we will refer to DiFUB by applying these two starting strategies as DiFUB+2*d*SWEEP*HdOut* and DiFUB+2*d*SWEEP*HdIn*.

7.3.1 Restricting to Undirected Graphs

If $G = (V, E)$ is an undirected graph and u is any vertex in V, $F_i^F(u)$, the *forward fringe* of u, coincides with $F_i^B(u)$, the *backward fringe*. By consequence $B_j^B(u) = B_j^F(u)$, and the Algorithm 20 in the case of undirected graph can be simplified as in Algorithm 24, as shown by [194]. Analogously to Theorem 7.3, it is possible to prove the following.

Algorithm 22: 2*d*SWEEP

Input: A strongly connected graph G, a vertex r
Output: A vertex and a lower bound for D

```
1  Run a forward BFS from r: let a_1 be the farthest vertex;
2  Run a backward BFS from a_1: let b_1 be the farthest vertex.
3  Run a backward BFS from r: let a_2 be the farthest vertex.
4  Run a forward BFS from a_2: let b_2 be the farthest vertex.
5  if ecc_B(a_1) > ecc_F(a_2) then
6  |  u ← the middle vertex between a_1 and b_1;
7  |  l ← ecc_B(a_1);
8  else
9  |  u ← the middle vertex between a_2 and b_2;
10 |  l ← ecc_F(a_2);
11 end
12 return u and l;
```

Theorem 7.5 *Algorithm* 24 *correctly computes the value of the diameter of* G.

In order to obtain a good starting vertex u in Algorithm 24, we can simplify 2*d*SWEEP Algorithm, as shown by Algorithm 23, that is the well known 2SWEEP Algorithm [19, 195].

1. Execute a forward breadth-first search starting from a vertex r: let a_1 be the farthest vertex.
2. Execute a forward breadth-first search starting from a_1: let b_1 be the farthest vertex.
3. Return the middle vertex between a_1 and b_1.

We will consider the variants called 2SWEEP*Hd*, in which r is the vertex with highest degree. Moreover we will refer to *i*FUB by applying this strategy to select the starting vertex as *i*FUB+2SWEEP*Hd* and to *i*FUB by starting from the vertex with the highest degree as *i*FUB*Hd*.

In a similar manner, Algorithm 21 can be simplified in order to deal with undirected graphs.

Algorithm 23: 2SWEEP

Input: A strongly connected graph G, a vertex r
Output: A vertex and a lower bound for D

```
1  Run a forward BFS from r: let a_1 be the farthest vertex;
2  Run a forward BFS from a_1: let b_1 be the farthest vertex.
3  u ← the middle vertex between a_1 and b_1;
4  l ← ecc_F(a_1);
5  return l;
```

Algorithm 24: *i*FUB

Input: An undirected connected graph G, a vertex u, a lower bound l for the diameter
Output: The diameter D

$i \leftarrow \mathrm{ecc}_F(u)$;
$lb \leftarrow \max\{\mathrm{ecc}_F(u), l\}$;
$ub \leftarrow 2i$;
while $ub - lb > 0$ **do**
  $lb \leftarrow \max\{lb, B_i^F(u)\}$;
  if $lb > 2(i-1)$ **then**
    return lb;
  else
    $ub \leftarrow 2(i-1)$;
  end
  $i \leftarrow i - 1$;
end
return lb;

Algorithm 25: D*i*FUB for weighted directed graphs

Input: A weighted directed strongly connected graph G, a vertex u, a lower bound for the diameter l
Output: The diameter D

Let $d_1 < d_2 < \cdots < d_h$ be the sequence of values d such that $F_d^{\mathrm{F}}(u) \neq \emptyset$ or $F_d^{\mathrm{B}}(u) \neq \emptyset$
$i \leftarrow h$;
$lb \leftarrow \max\{\mathrm{ecc}_F(u), ecc_B(u), l\}$;
$ub \leftarrow 2d_i$;
while $ub - lb > 0$ **do**
  $lb \leftarrow \max\{lb, B_{d_i}^B(u), B_{d_i}^F(u)\}$;
  if $lb > 2d_{i-1}$ **then**
    return lb;
  else
    $ub \leftarrow 2d_{i-1}$;
  end
  $i \leftarrow i - 1$;
end
return lb;

7.3.2 Generalizing to Weighted Graphs

Theorems 7.1 and 7.2 can be easily extended to the case of directed weighted graphs. Indeed, let T_u^F (respectively, T_u^B) denote the forward (respectively, backward) lightest path tree rooted at vertex u, computed, for instance, by means of the Dijkstra algorithm [196] in G (respectively, in the graph transposed of G). Moreover,

let $\mathrm{ecc}_F(u)$ (respectively, $\mathrm{ecc}_B(u)$) denote the weighted forward (respectively, backward) eccentricity of u, that is the weight of the longest path from (respectively, to) u to (respectively, from) one of the leaves of T_u^F (respectively, T_u^B). Finally, let $F_d^F(u)$ (respectively, $F_d^B(u)$) denote the set of vertices whose weighted distance from (respectively, to) u is equal to d: hence, $F_d^F(u) \neq \emptyset$ if and only if there exists at least one vertex x in T_u^F such that the weight of the path from u to x is equal to d, and $F_d^B(u) \neq \emptyset$ if and only if there exists at least one vertex x in T_u^B such that the weight of the path from x to u is equal to d.

Let $d_1, d_2, \ldots, d_h$ be the sequence of distinct values d such that $F_d^F(u) \neq \emptyset$ or $F_d^B(u) \neq \emptyset$ ordered in increasing order, that is, $d_1 < d_2 < \cdots < d_h$: note that $d_h = \max\{\mathrm{ecc}_F(u), \mathrm{ecc}_B(u)\}$. We then have the following two results, whose proofs are similar to the proofs of Theorems 7.1 and 7.2, respectively.

Theorem 7.6 *For any integer i with $1 < i \leq h$, for any integer k with $1 \leq k < i$, and for any vertex $x \in F_{d_{i-k}}^B(u)$ such that $\mathrm{ecc}_F(x) > 2d_{i-1}$, there exists $y \in F_{d_j}^F(u)$, for some $d_j \geq d_i$, such that $d(x, y) = \mathrm{ecc}_F(x)$. Moreover for any y such that $d(x, y) = \mathrm{ecc}_F(x)$, $y \in F_{d_j}^F(u)$, for some $d_j \geq d_i$.*

Theorem 7.7 *For any integer i with $1 < i \leq h$, for any integer k with $1 \leq k < i$, and for any vertex $x \in F_{d_{i-k}}^F(u)$ such that $\mathrm{ecc}_B(x) > 2d_{i-1}$, there exists $y \in F_{d_j}^B(u)$, for some $d_j \geq d_i$, such that $d(y, x) = \mathrm{ecc}_B(x)$. Moreover for any y such that $d(y, x) = \mathrm{ecc}_B(x)$, $y \in F_{d_j}^B(u)$, for some $d_j \geq d_i$.*

We can then appropriately modify the DiFUB algorithm in order to deal with directed weighted graphs. To this aim, we define

$$B_{d_i}^F(u) = \begin{cases} \max_{x \in F_{d_i}^F(u)} \mathrm{ecc}_B(x) & \text{if } F_{d_i}^F(u) \neq \emptyset \text{ and } d_i \leq \mathrm{ecc}_F(u), \\ 0 & \text{otherwise} \end{cases}$$

and

$$B_{d_j}^B(u) = \begin{cases} \max_{x \in F_{d_j}^B(u)} \mathrm{ecc}_F(x) & \text{if } F_{d_j}^B(u) \neq \emptyset \text{ and } d_j \leq \mathrm{ecc}_B(u), \\ 0 & \text{otherwise.} \end{cases}$$

The DiFUB algorithm for directed weighted graphs is then described in Algorithm 25. Analogously to Theorem 7.3, it is possible to prove the following.

Theorem 7.8 *Algorithm 25 correctly computes the value of the diameter of G.*

Observe that, in order to start the execution of the algorithm, we can also modify the 2dSweep algorithm by using single source lightest path algorithm executions instead of BFSes.

Algorithm 26: Computing the radius and enumerating all the radial vertices

Input: A graph G, directed or undirected, a pair of vertices x, y
Output: The radius R and C, the set of vertices whose forward eccentricity is R

```
1  π ← ordering of the vertices v ∈ V according to max{d(v, x), d(v, y)};
2  C = ∅;
3  ub ← n;
4  for i = 1 to n do
5  |  v ← π(i);
6  |  if ecc_F(v) < ub then
7  |  |  ub ← ecc_F(v);
8  |  |  C ← {v};
9  |  else
10 |  |  if ecc_F(v) = ub then
11 |  |  |  C ← C ∪ {v};
12 |  |  end
13 |  end
14 |  if i ≠ n then
15 |  |  u ← π(i + 1);
16 |  |  if ub < max{d(u, x), d(u, y)} then
17 |  |  |  return ub and C;
18 |  |  end
19 |  end
20 end
21 return ub and C;
```

7.4 Computing the Radius and Enumerating All the Radial Vertices

In the following, as before, we will assume that G is connected whenever G is undirected, or that G is strongly connected whenever G is directed. Algorithm 26 computes all the vertices whose eccentricity is equal to the radius. In particular, given in input a pair of vertices x, y, Algorithm 26 orders the vertices v according to $\max\{d(v, x), d(v, y)\}$ and scans their eccentricity. At each step, a lower bound on the eccentricities of the vertices still to process can be easily retrieved.

Remark 7.2 For any i, let u be $\pi(i + 1)$, then

$$ecc_F(z) \geq \max\{d(z, x), d(z, y)\} \geq \max\{d(u, x), d(u, y)\}$$

for any z such that $z = \pi(j)$, with $j \geq i + 1$.

For increasing values of i, we can compute the eccentricity of the vertex $v = \pi(i)$, and maintain the minimum eccentricity found, that is an upper bound ub for the radius. Remark 7.2 implies that $\max\{d(u, x), d(u, y)\}$ is a lower bound of the eccentricities

of the vertices to be processed. Thus if ub is less than this lower bound, we can stop the computation because it is not possible to decrease ub. In particular this implies that all the vertices, whose eccentricity has not yet been computed, have eccentricity strictly greater than R. Then the following result holds.

Lemma 7.1 *Algorithm* 26 *returns all and only the radial vertices.*

While scanning the eccentricities of the vertices, a list C, of the examined vertices whose eccentricity is equal to the current value of the upper bound, is maintained.

The performances of Algorithm 26 are strongly affected by the choice of the starting pair of vertices x, y. A good strategy is applying the 2*d*SWEEP algorithm and, referring to Algorithm 22, taking a_1 and b_2. In particular, in the following we will consider the 2*d*SWEEP*HdOut* and 2*d*SWEEP*HdIn*. We will refer to Algorithm 26 with starting strategies 2*d*SWEEP*HdOut* and 2*d*SWEEP*HdIn* as rad+2*d*SWEEP*HdIn* and rad+2*d*SWEEP*HdIn* respectively.

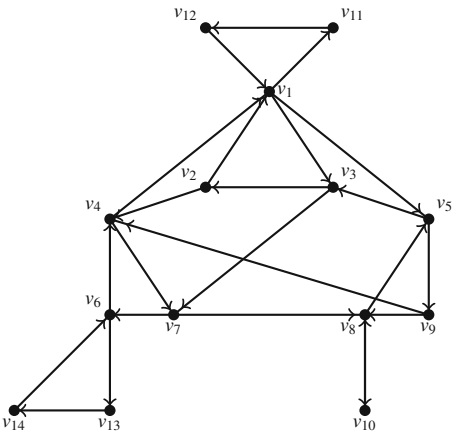
Observe that Algorithm 26 can be applied also in the case of undirected graphs. In this case, by applying 2SWEEP*Hd* Algorithm, referring to Algorithm 22, the vertices a_1 and b_1 can be used as starting vertices for Algorithm 26. In the following, we will refer to this Algorithm with this starting choice as rad+2*d*SWEEP*Hd*. Moreover, by using Dijkstra's Algorithm, instead of using the BFS Algorithm, the algorithm can be applied also in the case of weighted graphs.

7.5 Enumerating Diametral and Radial Vertices: An Example

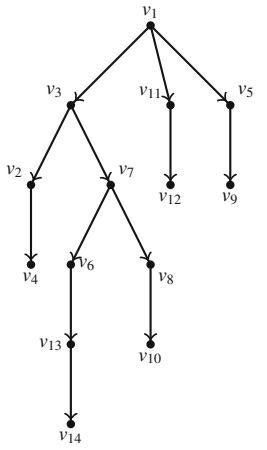
Let us consider the graph shown in the top part of Fig. 7.1, whose all pairwise distances, with the forward and backward eccentricities of all its vertices are shown. If we choose $u = v_1$, the corresponding two breadth-first search trees T_u^F and T_u^B are shown in the bottom part of the figure. From these two trees we can easily derive the forward and backward fringe sets, which are shown in the bottom part of the figure, together with $B_i^F(v_1)$, $B_i^B(v_1)$, $S_i^F(v_1)$, $T_i^F(v_1)$, $S_i^B(v_1)$, and $T_i^B(v_1)$.

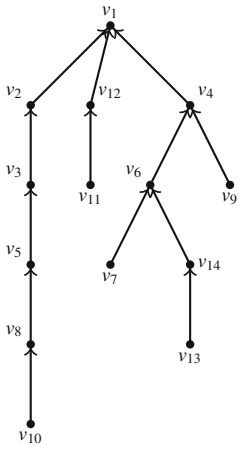
If we choose $i = 2$, $j = 3$, $x = v_6$, and $y = v_8$, then it is easy to verify, by inspecting the two BFSes trees, that we can go from v_6 to v_8 by first going up in $T_{v_1}^B$ (by means of two arcs) and then by going down in $T_{v_1}^F$ (by means of three arcs). Hence, as observed in Remark 7.1, $d(v_6, v_8) \leq 5$: indeed, $d(v_6, v_8) = 3$ (passing through v_4 and v_7). Moreover, if we choose $i = 2, k = 1$, and $x = v_4 \in F_1^B(v_1)$, then we have that $\mathrm{ecc}_F(v_4) = 4 > 2 = 2(i-1)$: Theorem 7.1 is in this case witnessed by vertex $y = v_2 \in F_2^F(v_1)$ (indeed, $d(v_4, v_2) = 3$).

Suppose we invoke Algorithm 21 with $u = v_1$ and $l = 0$. Before the execution of the **while** loop starts, the two variables i and lb are both set equal to $\max\{\mathrm{ecc}_F(v_1), \mathrm{ecc}_B(v_1)\} = 5$, while variable ub is set equal to $2i = 10$ and the sets DS and DT are both empty. Since $ub - lb = 5 > 0$, the algorithm enters the **while** loop



	v_1	v_2	v_3	v_4	v_5	v_6	v_7	v_8	v_9	v_{10}	v_{11}	v_{12}	v_{13}	v_{14}	ecc_F
v_1	0	2	1	3	1	3	2	3	2	4	1	2	4	5	5
v_2	1	0	2	1	2	3	2	3	3	4	2	3	4	5	5
v_3	2	1	0	2	3	2	1	2	4	3	3	4	3	4	4
v_4	1	3	2	0	2	2	1	2	3	3	2	3	3	4	4
v_5	3	2	1	2	0	3	2	2	1	3	4	5	4	5	5
v_6	2	4	3	1	3	0	2	3	4	4	3	4	1	2	4
v_7	3	4	3	2	2	1	0	1	3	2	4	5	2	3	5
v_8	4	3	2	3	1	4	3	0	2	1	5	6	5	6	6
v_9	2	4	3	1	2	3	2	1	0	2	3	4	4	5	5
v_{10}	5	4	3	4	2	5	4	1	3	0	6	7	6	7	7
v_{11}	2	4	3	5	3	5	4	5	4	6	0	1	6	7	7
v_{12}	1	3	2	4	2	4	3	4	3	5	2	0	5	6	6
v_{13}	4	6	5	3	5	2	4	5	6	6	5	6	0	1	6
v_{14}	3	5	4	2	4	1	3	4	5	5	4	5	2	0	5
ecc_B	5	6	5	5	5	5	4	5	6	6	6	7	6	7	





i	$F_i^F(v_1)$	$F_i^B(v_1)$	$B_i^F(v_1)$	$B_i^B(v_1)$	$S_i^F(v_1)$	$T_i^F(v_1)$	$S_i^B(v_1)$	$T_i^B(v_1)$
1	v_3, v_5, v_{11}	v_2, v_4, v_{12}	6	6	v_{11}	v_{10}	v_{12}	v_{14}
2	v_2, v_7, v_9, v_{12}	v_3, v_6, v_9, v_{11}	7	7	v_{12}	v_{10}	v_{11}	v_{14}
3	v_4, v_6, v_8	v_5, v_7, v_{14}	5	5	v_4, v_6, v_8	v_{10}, v_{11}, v_{13}	v_5, v_7, v_{14}	$v_2, v_9, v_{10}, v_{12}, v_{14}$
4	v_{10}, v_{13}	v_8, v_{13}	6	6	v_{10}, v_{13}	v_{10}, v_{11}, v_{13}	v_8, v_{13}	$v_2, v_9, v_{10}, v_{12}, v_{14}$
5	v_{14}	v_{10}	7	7	v_{14}	v_{10}, v_{11}	v_{10}	v_{12}, v_{14}

Fig. 7.1 A strongly connected graph with the corresponding all pairwise distances, forward and backward eccentricities and BFSes trees rooted at v_1, and the fringe set properties

with $i = 5$. Since, $B_5^B(v_1) = 7 > lb$, lb is set to 7 and DS and DT are respectively set to $\{v_{10}\}$ and $\{v_{12}, v_{14}\}$. Then since $B_5^F(v_1) = 7$ is equal to lb, $DS = \{v_{10}\} \cup \{v_{10}, v_{11}\}$ (v_{11} is added to DS) and $DT = \{v_{12}, v_{14}\} \cup \{v_{14}\}$ (DT does not change). Since $lb = 7 < 8 = 2(i-1)$, the algorithm sets ub equal to 8 and performs another iteration with $i = 4$. Since $B_4^F(v_1) = B_4^B(v_1) = 6$ and $6 < 7 = lb$ the lower bound lb is not improved and the sets DS and DT are not modified. Since $lb = 7 > 2(i-1) = 6$ the algorithm returns: $lb = 7$ is the diameter of the graph, $DS = \{v_{10}, v_{11}\}$ is the set of all the diametral sources (indeed $ecc_F(v_{10}) = ecc_F(v_{11}) = 7$), and $\{v_{12}, v_{14}\}$ is the set of all the diametral targets (indeed $ecc_B(v_{12}) = ecc_B(v_{14}) = 7$).

Finally suppose we invoke Algorithm 26, with input the pair of vertices v_1, v_{10}. We define the sets C_i as $\{u : \max\{d(u, v_1), d(u, v_{10})\} = i\}$ and for any i we show the corresponding C_i in the following.

i	$C_i = \{u : \max\{d(u, v_1), d(u, v_{10})\} = i\}$
1	$\emptyset$
2	v_9
3	v_3, v_4, v_5, v_7
4	v_1, v_2, v_6, v_8
5	v_{10}, v_{12}, v_{14}
6	v_{11}, v_{13}

Thus Algorithm 26 order the vertices as follows

$$\langle v_9, v_3, v_4, v_5, v_7, v_1, v_2, v_6, v_8, v_{10}, v_{12}, v_{14}, v_{11}, v_{13}\rangle.$$

ub is set to n and C is empty. In the first iteration we have $v = v_9$ and since $ecc_F(v_9) = 5$, ub is set to 5 and $C = \{v_9\}$. Since $ub = 5 > \max\{d(v_3, v_1), d(v_3, v_{10})\} = 3$, where v_3 is the successor of v_9 in the order, the algorithm performs a new iteration. In this latter iteration $v = v_3$, ub is improved and set to $ecc_F(v_3) = 4$ and $C = \{v_3\}$. Since $ub = 4 > \max\{d(v_4, v_1), d(v_4, v_{10})\} = 3$, the algorithm does not stop and considers as new v the vertex v_4. Since $ecc_F(v_4) = 4$, v_4 is added to C, so that $C = \{v_3, v_4\}$. Since $ub = 4 > \max\{d(v_5, v_1), d(v_5, v_{10})\} = 3$, the algorithm performs an iteration by using v_5 as v: in this case ub is not improved and C does not change. The same happens by considering as v the vertices v_7, v_1, and v_2. When considering as v the vertex v_6, since $ecc_F(v_6) = 4 = ub$, ub is not improved and v_6 is added to C. Since $ub = 4 = \max\{d(v_8, v_1), d(v_8, v_{10})\} = 4$, a new iteration is performed, where $v = v_8$. At the end of this latter iteration, since $ecc_F(v_8) = 6 > ub$, ub is not improved and C is not changed; since $ub = 4 < \max\{d(v_{10}, v_1), d(v_{10}, v_{10})\} = 5$, the algorithm stops. Indeed all the vertices after v_{10} have ecc_F at least $\max\{d(v_{10}, v_1), d(v_{10}, v_{10})\} = 5$. Thus $ub = 4$ is the radius of the graph, and $C = \{v_3, v_4, v_6\}$ is the set of radial vertices, since $ecc_F(v_3) = ecc_F(v_4) = ecc_F(v_6) = 4$.

7.6 *Ad Hoc* Bad Cases

There exist graphs, such as the graph shown in Fig. 7.2 and other graphs available at [197], where our algorithms use $\Theta(n)$ BFSes. These graphs are characterized by an extreme regularity: the BFS trees at their vertices are very similar, with the radius and the diameter values very close, namely, $R \approx D$, and all vertices have close eccentricity. Figure 7.2 show an example in which *i*FUB performs $\Theta(n)$ BFSes even if the number of diametral vertices is constant, while Fig. 7.4 show an example in which Algorithm 26 performs $\Theta(n)$ BFSes even if the number of radial vertices is constant.

Whenever R is close to D, about $D/2$ iterations will always be executed in the *i*FUB algorithm, so that the number of visits performed by *i*FUB is more likely also close to n, even if the number of diametral vertices is constant. Thus in the case of these graphs, the complexity of *i*FUB is $\Theta(nm)$. For instance, a cycle with n vertices (n odd), $v_1, \ldots, v_n$, connected to a vertex v_{n+1} by the edge (v_1, v_{n+1}) has diameter $\frac{n+1}{2}$. Any of its vertices has the same BFS tree, whose height is $\frac{n-1}{2}$, except for v_{n+1}, $v_{\frac{n+1}{2}}$, and $v_{\frac{n+1}{2}+1}$ whose eccentricity is $\frac{n+1}{2}$, (see Fig. 7.2). Thus, referring to Algorithm 20 and Algorithm 21, by starting from any vertex with eccentricity $\frac{n-1}{2}$, *i*FUB repeats its loop until $2(i-1) \geq \frac{n-1}{2}+1$, that is $i \geq \frac{n+5}{4}$, and stops the first time that $2(i-1) < \frac{n-1}{2}+1$, for a total number of iterations equal to $\frac{n-1}{2} - \frac{n+5}{4} + 2 = \frac{n+1}{4}$. Since each level has at least two vertices, the number of BFSes performed by *i*FUB is thus linear and greater than $\frac{n+3}{2}$. Similar result can be found if Algorithm 20 and Algorithm 21 start from v_{n+1}, $v_{\frac{n+1}{2}}$, or $v_{\frac{n+1}{2}+1}$. These bad undirected graphs for *i*FUB can be easily adapted to build bad directed graphs for D*i*FUB, by replacing each edge by two opposite arcs.

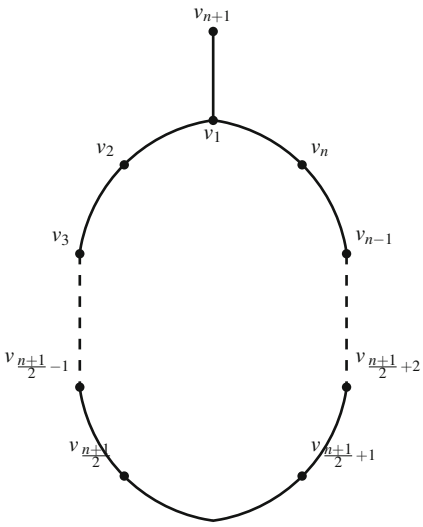


Fig. 7.2 A bad network for Algorithm 21. By replacing each edge with two opposite arcs, this graph becomes a bad network for D*i*FUB algorithm

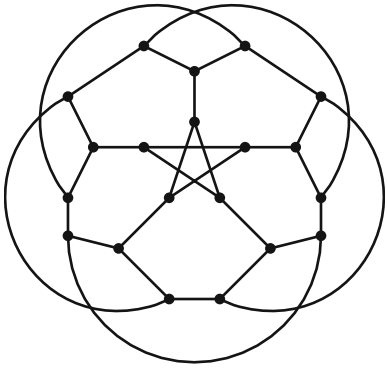


Fig. 7.3 A bad network for Algorithms 20 and 26 with small diameter. By replacing each edge with two opposite arcs, this graph becomes a bad network for the directed versions of the algorithms

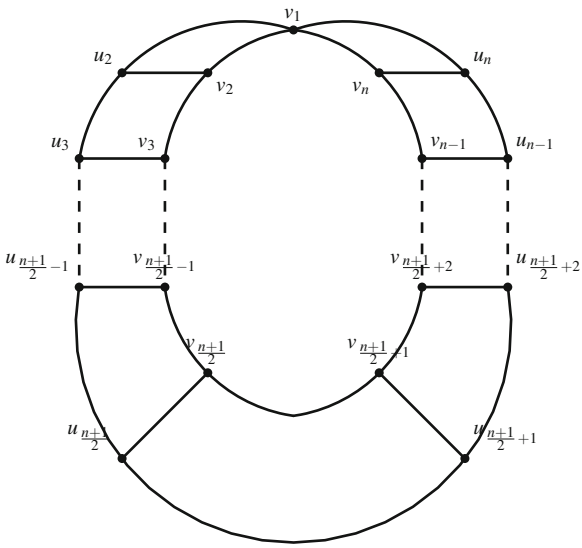


Fig. 7.4 A bad network for Algorithm 26. By replacing each edge with two opposite arcs, this graph becomes a bad network for the directed version of the algorithm

Observe that this graph has linear diameter. However there are graphs, like the one shown in Fig. 7.3 and their generalizations in [197], whose diameter is small, and for which Algorithm 20 uses a linear number of visits.

In the graph shown in Fig. 7.4, that is composed by two cycles $v_1, \ldots, v_n$ and $v_1, u_2, \ldots u_n$, such that for any i, with $2 \leq i \leq n$, there is an edge between u_i and v_i, every vertex have eccentricity equal to the diameter $D = \frac{n+1}{2}$, except for v_1, whose eccentricity is equal to the radius $R = D - 1$. If the starting pair of vertices for Algorithm 26 is $u_2, v_{\frac{n+1}{2}+1}$, then v_1 can be one of the last vertex to be processed, so that the number of visits performed by Algorithm 26 is linear. Once again, this bad undirected graph can be easily adapted to build an example of bad directed graph, by replacing each edge by two opposite arcs.

7.7 Experiments

We collected several real-world directed or undirected graphs, which have been chosen in order to cover the largest possible set of network typologies. An important feature is that almost all graphs in our dataset are sparse (that is, $m = O(n)$). Note that, in the case of several of these graphs, the diameter value was still unknown.

Our computing platform is a machine with a Pentium Quad-Core CPU (Intel(R) Xeon(R) E5405 @ 2.00GHz), with a 10GB shared memory. The operating system is a Ubuntu GNU/Linux 12.04.1, with a Linux version 3.2.0-34 and gcc version 4.6.3. For each network we report number of vertices and arcs/edges in the original graph and in the biggest (strongly) connected component. For this latter we have computed the diameter D and the radius R, together with the diametral sources, the diametral targets, and the radial vertices.
The code and the data set are available at `amici.dsi.unifi.it/lasagne/`.

In Sects. 7.7.1 and 7.7.2 we will describe our experiments for directed and undirected graphs, respectively. In Sect. 7.7.3 we will summarize our results.

7.7.1 Directed Graphs

In the case of directed graphs, we have experimented Algorithm 21 by using several ways of choosing the starting vertex.

- DiFUB*HdOut*. DiFUB by starting from the vertex with highest out-degree.
- DiFUB*HdIn*. DiFUB by starting from the vertex with highest in-degree.
- DiFUB+2*d*SWEEP*HdOut*. Apply 2*d*SWEEP*HdOut*, Algorithm 22 by starting from the vertex with highest out-degree, and obtain the vertex u; apply DiFUB by starting from u.
- DiFUB+2*d*SWEEP*HdIn*. Apply 2*d*SWEEP*HdIn*, Algorithm 22 by starting from the vertex with highest in-degree, and obtain the vertex u; apply DiFUB by starting from u.

Observe that each experiment is deterministic, since ties between vertices are broken by considering the vertex whose index is minimum.[1]

Moreover we have experimented Algorithm 26 by considering two ways of selecting the starting pair of vertices.

- rad+2*d*SWEEP*HdOut*. Apply 2*d*SWEEP*HdOut*, Algorithm 22 by starting from the vertex with highest out-degree, and obtain the vertex u; apply Algorithm 26 by starting from u.
- rad+2*d*SWEEP*HdIn*. Apply 2*d*SWEEP*HdIn*, Algorithm 22 by starting from the vertex with highest in-degree, and obtain the vertex u; apply Algorithm 26 by starting from u.

[1] Ties are not frequent and no substantial differences have been observed by breaking ties in different way.

In the following we will describe the results of our experiments for each category of networks.

- **Biological Networks (from [49, 85])**. These are metabolic networks, in particular bipartite networks, compound networks, and reaction networks, taken from [49, 85]. In particular we have considered only the 76 biological sources whose number of vertices of the strongly connected component of the bipartite network is greater than 500. In Bipartite Metabolic Networks, all the DiFUB methods use a number of visits less than $10\%n$ for more than 66 networks. rad+2dSWEEP*HdOut* and rad+2dSWEEP*HdOut* use a number of visits less than $15\%n$ for 68 networks. In Compound Metabolic Networks, all the DiFUB methods use a number of visits less than $10\%n$ for more than 61 networks, while the radius methods use a number of visits less than $16\%n$ for more than 66 networks. In Reaction Metabolic Networks all the DiFUB methods except DiFUB+2dSWEEP*HdOut*, use a number of visits less than $10\%n$ for more than 61 networks, while for DiFUB+2dSWEEP*HdOut* this is true just for 38 networks. rad+2dSWEEP*HdOut* and rad+2dSWEEP*HdOut* use a number of visits less than $15\%n$ for 67 and 38 networks respectively.
- **Directed Social Networks (from [198, 199])**. This class of networks includes for instance a *who-trust-whom* online social network of the general consumer review site `Epinions.com`, the social network of `LiveJournal`, that is a free online community allowing members to maintain journals, individual and group blogs, and allowing people to declare which other members are their friends, and the social network of `Slashdot`, that is a technology-related news website that allows users to tag each other as friends or foes. Moreover there is the *who-votes-on-whom network* Wikipedia network: indeed Wikipedia is a free encyclopedia written collaboratively by volunteers and in order for a user to become an administrator a request for adminship is issued and the Wikipedia community via a public discussion or a vote decides who to promote to adminship [198]. In this class all the DiFUB algorithms use always less visits than 0.1 % of the number of vertices of the largest strongly connected component, n, except for `soc-Slashdot0811` and `WikiVote` in which DiFUB uses respectively less than $1\%n$ and $2\%n$. The computation of radial vertices has used always a number of visits less than $1\%n$ except for `soc-Slashdot0902`, `soc-Slashdot0811`, and `wiki-Vote`, in which the number of visits is less than $25\%n$. For the graph `ljournal-2008` we were not able to complete our experiments in the case of rad+2dSWEEP*HdOut* and rad+2dSWEEP*HdIn* (see Sect. 7.7.3).
- **Web Networks (from [198, 199])**. Vertices represent pages and arcs represent hyperlinks between them. The pages considered are restricted to the one belonging to `berkely.edu`, and `stanford.edu` domains, the `nd.edu` domain, that is University of Notre Dame (released by Albert Barabási), the pages considered in 2002 by Google as a part of a Google Programming Contest, the .it domain, the Italian CNR domain, the `.in` and `.indochina` domains (crawled by the Nagaoka University of Technology), the `.eu` domain (collected for the DELIS project, a collection of web graphs by taking snapshots at a monthly rate focussing on the `.uk` domain). Moreover there are the snapshots performed by UbiCrawler

of the `.uk` domain in 2002 and 2005, of the `.sk` domain in 2005, and one aimed at countries whose web sites could contain pages written in Arabic in 2005. Finally we have considered the snapshot obtained by the WebBase crawler in 2001. In this class of networks DiFUB has used a number of visits always less than $0.1\%n$, except when DiFUB*HdOut* and DiFUB*HdIn* are applied to: `in-2004`, in which the number of visits is less than $4\%n$, `uk-2007-05@100000`, in which the number of visits is about $75\%n$, and `uk-2007-05@1000000`, in which the number of visits is less than $0.5\%n$. In order to compute all the radial vertices, our algorithm takes always less than $1\%n$ number of visits, except for the graph `web-Stanford`, in which it has used about than $10\%n$ visits. For the biggest graphs, i.e. the ones with more than 3.5 million of vertices, we were not able to complete our experiments concerning the radius: indeed the time needed on our platform for each BFS is greater than 13 s (as for `indochina-2004`), so that, even by performing just $3\%n$ visits, more than two weeks are needed in order to complete one single experiment (see Sect. 7.7.3).

- **Citation Networks (from [198])**. Arxiv Hep-Ph (high energy physics phenomenology) and Hep-Th (high energy physics theory) citation graphs are from the e-print arXiv and covers all the citations from January 1993 to April 2003 in their respective categories. If a paper i cites paper j, the graph contains an arc from i to j. Both DiFUB+2dSWEEP*HdOut* and DiFUB+2dSWEEP*HdIn* have always used a number of visits less than $1\%n$ in both the graphs. DiFUB*HdOut* and DiFUB*HdIn* have used a number of visits respectively about $15\%n$ and $75\%n$ for cit-HepPh, and both less than $5\%n$ for `cit-HepTh`. The computation of radial vertices instead has required always less than $1.5\%n$ number of visits.
- **Communication Networks (from [198, 199])**. In this class of networks, given a set of email messages, each vertex corresponds to an email address, and there is an arc between vertices i and j, if i sent at least one message to j. This class of networks include a network generated using email data from a large European research institution, Enron email communication network covering all the email communication within a dataset of around half million emails (this data was originally made public, and posted to the web, by the Federal Energy Regulatory Commission during its investigation). Moreover in the `wiki-Talk` network, the vertices represent Wikipedia users and an arc from vertex i to vertex j represents that user i at least once edited a talk page of user j. In all these graphs DiFUB has used less than or about $1\%n$ visits to list all the diametral vertices. The computation of all the radial vertices has required always a number of visits between $10\%n$ and $15\%n$, except for `wiki-Talk`.
- **P2P Networks (from [198])**. A sequence of snapshots of the `Gnutella` peer-to-peer file sharing network from August 2002. Vertices represent hosts in the `Gnutella` network topology and edges represent connections between the `Gnutella` hosts. In these networks DiFUB uses less than $5\%n$ visits, except for `p2p-Gnutella09` (in which a number of visits between $5\%n$ and $12\%n$ are used) `p2p-Gnutella30` (in which a number of visits between $3\%n$ and $15\%n$ are used), `p2p-Gnutella08` (in which DiFUB*HdOut* uses about $24\%n$ visits), and `p2p-Gnutella25`, (in which DiFUB*HdIn* uses about $19\%n$ visits).

The number of visits required for the computation of the radial vertices ranges between $0.4\%n$ and about $5\%n$, except when rad+2dSWEEP*HdIn* is applied to `p2p-Gnutella08` and `p2p-Gnutella30`, requiring about $12\%n$ visits.

- **Product co-purchasing Networks (from [198, 199])**. These networks were collected by crawling Amazon website and are based on *Customers Who Bought This Item Also Bought* feature of the Amazon website, so that if a product i is frequently co-purchased with product j, the graph contains an arc from i to j. For these networks D*i*FUB has used almost always less than $0.05\%n$ visits, except for `amazon0312` and `amazon0302` in which the number of visits used ranges between $0.06\%n$ and $1.32\%n$. The computation of the radial vertices requires instead a number of visits in between $0.01\%n$ and $0.72\%n$.
- **Word Association Networks (from [199])**. The Free Word Association Norms Network is a directed graph describing the results of an experiment of free word association performed by more than 6000 participants in the United States: its vertices correspond to words and arcs represent a cue-target pair and an arc from x to y means that the word y was output by some of the participants based on the stimulus x [199] . The word association network seems to be a real world negative example for our algorithms: D*i*FUB uses between $20\%n$ and $70\%n$ visits in all the strategies while the computation of the radial vertices requires about $45\%n$ visits.

7.7.2 Undirected Graphs

The effectiveness of Algorithm 24 has been experimentally proved in [20, 21, 194] by using several ways of choosing the starting vertex.

- *Random selection.* By picking it uniformly at random.
- *Degree selection.* By choosing a vertex with the highest degree.
- *4-Sweep selection.* By using the 4SWEEP method, that is an evolution of 2SWEEP method using four BFSes. Let r_1 be a vertex in V, let a_1 be one of the farthest vertices from r_1, and let b_1 be one of the farthest vertices from a_1. If r_2 is the vertex halfway between a_1 and b_1, then we define analogously a_2 and b_2. The vertex u is then defined as the middle vertex of the path between a_2 and b_2. In particular, two different ways of selecting vertex r_1 have been considered: one method chooses r_1 uniformly at random, while the other method chooses r_1 as a vertex with the highest degree.

By using these latter two selection strategies, it has been shown that in almost any graph with more than 10,000 vertices of the considered dataset, the number of visits is always much less than 0.1% of the number of all vertices in the largest connected component, except for the road networks (in which the number of executions of the shortest path algorithm performed is less than 10%) and Erdős-Rényi graphs. Instead it has been shown that it is not convenient to run *i*FUB by starting from random vertices, since sometimes the number of performed visits is high with respect to the number of vertices.

In the following we will describe the experimental evaluation of the performances of *i*FUB*Hd* and *i*FUB+2SWEEP*Hd* when used not just to compute the diameter but also to enumerate the diametral vertices.

Thus we have experimented Algorithm 21, simplified as shown by Algorithm 24 for undirected graphs, by using the following ways of choosing the starting vertex.

- *i*FUB*Hd*. *i*FUB by starting from the vertex with highest degree.
- *i*FUB+2SWEEP*Hd*. Apply 2SWEEP*Hd*, Algorithm 23 by starting from the vertex with highest degree, and obtain the vertex u; apply *i*FUB by starting from u.

Moreover we have experimented Algorithm 26 by using:

- rad+2*d*SWEEP*Hd*. Apply 2SWEEP*Hd*, Algorithm 23, by starting from the vertex with highest degree, and obtain the vertex u; apply Algorithm 26 by starting from u.

Observe that several networks that we have tested are not really naturally undirected since the relationship that they represent is implicitly not symmetrical. However since they are released by the owner companies or the owner universities in undirected format, we have catalogued them in this section.

- **Protein-Protein Interaction Networks (from [200–206])**. In these networks *i*FUB has almost always used a number of visits in between $0.1\%n$ and $1\%n$. In the graph `iPfam` it has used about $6\%n$ visits. For the computation of the radial vertices the number of visits has been always less than $3\%n$ except for `psimap` ($6\%n$), `string` ($35\%n$), `interdom` ($43\%n$).
- **Collaboration networks (from [198, 200, 205, 207–211])**. This class includes the graph of movie actors using data from IMDB (Internet Movies DataBase), whose vertices are actors, and two actors are linked by an edge whenever they collaborated in a movie. Moreover there are the graphs of co-authorship (if an author i co-authored a paper with author j, the graph contains an edge from i to j), based on DBLP, MathSciNet [212], Arxiv Astro Physics, Arxiv Condensed Matter, Arxiv General Relativity, Arxiv High Energy Physics, Arxiv High Energy Physics Theory e-print categories arXiv, and the original Condensed Matter section of arXiv E-Print Archive between 1995 and 1999 used by Newman in [127]. More peculiar are the networks `Eva`, in which there is an edge (x, y) from company x to company y whether the company x is an owner of company y or vice versa [213], `Advogato`, a research testbed for testing attack-resistant trust metrics [208], the network of the collaborations among Jazz musicians [214], and the network of users of the *Pretty-Good-Privacy* algorithm for secure information interchange [215]. Observe that if the collaboration involves k collaborators, this generates a completely connected subgraph on k vertices. In these networks *i*FUB has used less than or about $1\%n$ visits except for the smallest networks, `jazz`, `eva`, `geom`, `Newman-Cond_mat`, and `PGPgiantcompo`. The number of visits required for the computation of the radial vertices is always less tha $5\%n$, except for `jazz`, `advogato`, and `Newman-Cond_mat`.
- **Undirected Social Networks (from [205, 208, 216])**. These networks include *MySpace* and *Flickr* friendship, the membership of *Yahoo!* users to *Yahoo!* groups,

and one network used to test trust metrics. The number of visits required to compute all the diametral and radial vertices is less than $0.06\%n$, except for `myspace` in which the diametral and radial vertices are computed with less than $2.2\%n$ visits.

- **Undirected Communication Networks (from [216, 217])**. There is an edge between two vertices whether there has been a communication between them. In these networks *i*FUB+2SWEEP*Hd* uses always less than $1\%n$ visits, while *i*FUB*Hd* uses almost always less than $5\%n$ visits, except for `d21.txt` and `d02.txt`. The computation of the radial vertices has required always less than $1\%n$ visits except for `halfyearA` and `oneyearA` (at most $7\%n$ visits).
- **Autonomous Systems Networks (from [198, 200])**. These networks are `as-skitter` and `itdk0304_rlinks_undirected`, that are Internet topology graphs from traceroutes run respectively in 2004 and in 2005. The number of visits required by *i*FUB is always less than $0.02\%n$, while the radial vertices can be computed with at most $0.36\%n$ visits.
- **Road Networks (from [198])**. These graphs are the road networks of California, Pennsylvania, and Texas. Intersections and endpoints are represented by vertices and the roads connecting these intersections or road endpoints are represented by edges [198]. The completed experiments are about rad+2*d*SWEEP*Hd*: for `roadNet-PA` and `roadNet-TX` the number of visits has been less than $0.2\%n$, while for `roadNet-CA` the number of visits has been less than $15\%n$. The values of the diameters have been computed by applying Algorithm 20 (see [21]).
- **Word Adjacency Networks (from [207, 216, 218])**. In these networks, the vertices are words and there is an edge between x and y whether x appears close to y in at least one phrase of the considered book or web pages. The computation of the diametral and radial vertices requires almost always at most $1.5\%n$, except for the networks `eatRS` and `eatSR`, and when rad+2*d*SWEEP*Hd* is applied to `japaneseBookInter_st`.

7.7.3 Overall Results

In the following we resume the results of our experiments, grouping them by number of visits v, and showing for any method the number of networks in which it has performed v visits.

In the case of directed graphs, in order to compute the diametral vertices, D*i*FUB+2*d*SWEEP*HdOut* and D*i*FUB+2*d*SWEEP*HdIn* seems to be the more promising strategies. No substantial differences have been observed between the methods rad+2*d*SWEEP*HdOut* and rad+2*d*SWEEP*HdIn* (Table 7.1).

In Fig. 7.5a we report the number of visits performed by D*i*FUB*HdOut*, D*i*FUB*HdIn*, D*i*FUB+2*d*SWEEP*HdOut*, D*i*FUB+2*d*SWEEP*HdIn* to compute the diameter and the diametral vertices, as a function of the number of vertices. Analogously in Fig. 7.5b we report the number of visits performed to compute the radius and the radial vertices using rad+2*d*SWEEP*HdOut*, rad+2*d*SWEEP*HdIn*, as a function of the number

Table 7.1 For each method and for each v, the number of networks in which the method performs v visits to compute diameter or radius in directed graphs

v	# Networks in which the method performs v visits					
	Methods					
	Diameter				Radius	
	DiFUB *HdOut*	DiFUB *HdIn*	DiFUB 2*d*SWEEP *HdOut*	DiFUB 2*d*SWEEP *HdIn*	rad 2*d*SWEEP *HdOut*	rad 2*d*SWEEP *HdIn*
$v \leq 0.01\%n$	10	13	14	14	1	3
$0.01\%n < v \leq 0.1\%n$	13	13	16	16	9	7
$0.1\%n < v \leq 1\%n$	10	8	8	7	14	14
$1\%n < v \leq 10\%n$	9	9	7	8	9	7
$10\%n < v$	5	4	2	2	6	8

of vertices. In particular for each one of the directed graphs presented, having x vertices, in which a method performs y visits, we draw in position (x, y) the symbol corresponding to the method.

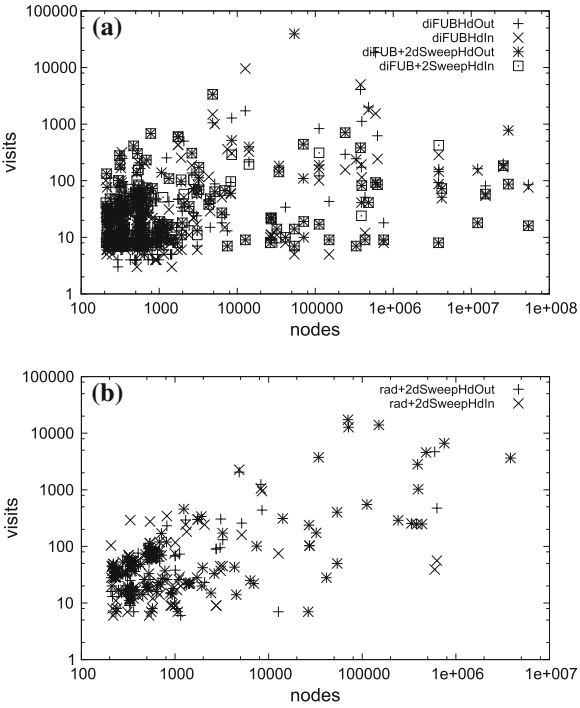


Fig. 7.5 Visits performed by rad+2*d*SWEEP*HdOut*, rad+2*d*SWEEP*HdIn* to compute the radius and the radial vertices, and visits performed by DiFUB*HdOut*, DiFUB*HdIn*, DiFUB+2*d*SWEEP*HdOut*, DiFUB+2*d*SWEEP*HdIn* to compute the diameter and the diametral vertices as a function of the number of vertices. For each one of the directed graphs presented, with x vertices, in which a method performs y visits, we draw in position (x, y) the symbol corresponding to the method

Observe that in the case of Fig. 7.5a, when the number of vertices increases, no increase can be detected in the number of visits. We argue that our methods perform a constant number of visits in practice. It is worth observing that the maximum number of visits correspond to the application of D*i*FUB*HdOut* and D*i*FUB*HdIn* to `uk-2007-05@100000`: we argue that the in-degree and the out-degree are not good centrality measures for this graph: indeed for the same graph, D*i*FUB is effective by starting from other vertices.

In the case of Fig. 7.5b, when the number of vertices increases, the number of visits slightly increases, so that whenever a graph have more than 3.5 millions of vertices, it seems that 100 thousands of visits (i.e. $2.8\%n$) are required and if each visit takes more than 13 s, at least 15 days are required to conclude one experiment. This is the case of the following 7 Networks: `uk-2002`, `indochina-2004`, `it-2004`, `arabic-2005`, `uk-2005`, `webbase-2001`, and `ljournal-2008`, for which we were not able to conclude our experiments concerning radius and radial vertices. Observe that according to our statistics, whenever the exhaustive method would be applied, for these graphs at least 526 days would be required to conclude one experiment. In these cases, the methods using external memory seem to be a promising alternative [219, 220].

In the case of undirected graphs, *i*FUB+2SWEEP*Hd* seems to be more stable than *i*FUB*Hd* in order to compute the diametral vertices. Indeed even if there are some networks in which *i*FUB*Hd* performs less than $0.01\%n$ visits and *i*FUB+2SWEEP*Hd* performs slightly more than $0.01\%n$ visits, there are networks in which *i*FUB*Hd* performs more than $1\%n$ visits and *i*FUB+2SWEEP*Hd* performs much less visits (Table 7.2).

In Fig. 7.6a we report the number of visits performed to compute the diameter and the diametral vertices by using *i*FUB*Hd* and *i*FUB+2SWEEP*Hd*, as a function of the number of vertices. Analogously in Fig. 7.6b we report the number of visits performed to compute the radius and the radial vertices by using rad+2*d*SWEEP*Hd*, as a function of the number of vertices. In particular for each one of the undirected

Table 7.2 For each method and for each v, the number of networks in which the method performs v visits to compute diameter or radius in undirected graphs

v	# Networks in which the method performs v visits		
	Methods		
	Diameter		Radius
	*i*FUB *Hd*	*i*FUB 2SWEEP *Hd*	rad 2SWEEP *Hd*
$v \leq 0.01\%n$	13	9	5
$0.01\%n < v \leq 0.1\%n$	13	27	21
$0.1\%n < v \leq 1\%n$	26	24	26
$1\%n < v \leq 10\%n$	17	9	14
$10\%n < v$	4	4	7

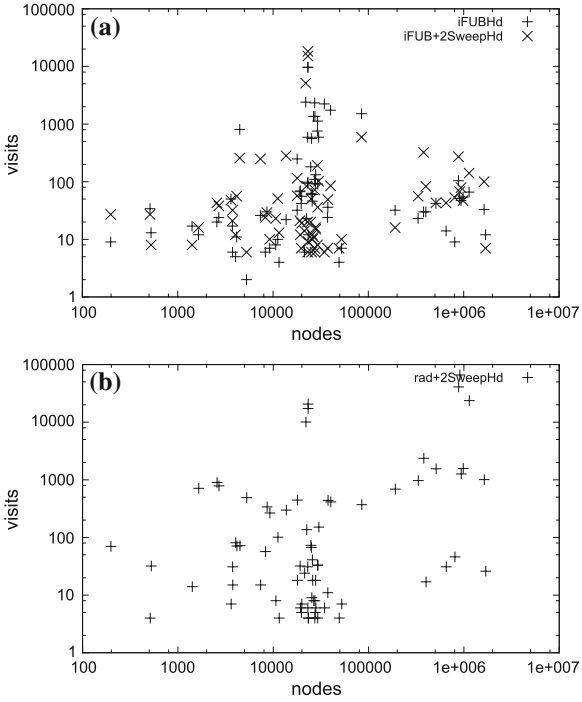


Fig. 7.6 Visits performed by *i*FUB*Hd* and *i*FUB+2SWEEP*Hd* to compute the diameter and the diametral vertices, and visits performed by rad+2*d*SWEEP*Hd* to compute the radius and the radial vertices as a function of the number of vertices. For each one of the directed graphs presented, with x vertices, in which a method performs y visits, we draw in position (x, y) the symbol corresponding to the method

graphs presented, having x vertices, in which a method performs y visits, we draw in position (x, y) the symbol corresponding to the method.

Observe that once again in the case of Fig. 7.6a, when the number of vertices increases, no increase can be detected in the number of visits. We argue that our methods perform a constant number of visits in practice. We conjecture that the impossibility of concluding our experiments in the case of *i*FUB for `roadNet-CA`, `roadNet-PA` and `roadNet-TX` is due to unidentified special topological properties of these graphs.

Once again in the case of Fig. 7.6b, when the number of vertices increases, the number of visits very slightly increases, but in this case this does not hinder the central vertex computations.

Finally we would like to point out that the lower bounds provided by the 2SWEEP or the 4SWEEP methods turn out to be, in practice, almost always tight: however, there is, in theory, no guarantee about the quality of the approximation. For this reason, some methods have been proposed in [19, 195] in order to find an upper bound on the diameter which bounds the absolute error or even validates the tightness of the lower bound. *i*FUBis a generalization of [19], meaning that when the second one stops because the diameter is found, also the first one stops: see [19] for a comparison with these upper bounding techniques. Moreover, recently and independently from this work, a new algorithm to compute the diameter of large real-world networks has been proposed in [141]: see [21] for a comparison with this work.

7.8 Conclusion and Open Problems

In the previous sections we have described and experimented new algorithms for computing the diameter and radius of directed and undirected (weighted) graphs, together with all the diametral and radial vertices. Even though these algorithms have $O(nm)$ time complexity in the worst case, our experiments suggest that their execution for real-world networks requires time $O(m)$ in the case of the diameter and almost $O(m)$ in the case of the radius.

The computation of the radius with our algorithm is affected by the choice of the starting vertices x, y so that the best performances are achieved whenever x and y are both diametral targets. The performance of DiFUB depends on the choice of the starting vertex u (indeed, it could be interesting to experimentally analyse its behaviour depending on this choice). Ideally, u should be such that the maximum between the forward and the backward eccentricity of u should be close to the $\min_{v \in V}\{\max\{ecc_F(v), ecc_B(v)\}\}$. Surprisingly, we have observed that in the case of real-world graphs, this value is close to the minimum possible, that is $D/2$. This peculiar structural property affects the performance of our algorithm: in these cases, the upper bound on the iterations is minimum and equal to $R - D/2 + 1$.

The main fundamental questions are now the following. Why our methods, both in the directed and in the undirected version, are so effective in finding the radius, the diameter, and vertices with high and low eccentricity? Which one is the topological underlying property that can lead us to these results? Why real world graphs exhibit this property? Some progress has been done by [221], but still a lot has to be done. Finally, it could be interesting to analyse a parallel implementation of the DiFUB algorithm. Indeed, the eccentricities of the vertices belonging to the same fringe set can be computed in parallel. Moreover, a variety of parallel BFS algorithms have been explored in the literature and can be integrated in the implementation of our algorithm.

Both the algorithms for diameter and radius described in this chapter have been very recently improved: we invite the interested reader to see [222].

Chapter 8
Conclusions

In this work we have resumed the main schema to design enumeration algorithms. We have seen that an useful application of enumeration algorithms is biological network analysis since biological network models introduce several biases: arc dependencies are neglected and underlying hyper-graph behaviours are forced in simple graph representations to avoid intractability. Moreover, regulatory interactions between all the biological networks are omitted, even if none of the different biological layers is truly isolated. Last but not least, the dynamical behaviours of biological networks are often not considered: indeed most of the currently available biological network reconstructions are potential networks, where all the possible connections are indicated, even if edges/arcs and vertices are hardly present all together at the same time. In this scenario, we have seen that very often enumeration algorithms can be helpful so that the solutions of a problem can be checked *a posteriori* by biologists.

The several techniques to design efficient enumeration algorithms include: brute force approaches, producing solutions and checking whether these are already been generated; approaches guaranteeing a bound on the time needed to produce two consecutive solutions; approaches guaranteeing a bound relating the overall time to enumerate all the solutions and the number of solutions (or their size). Hence, we have seen a new research example for each of these categories, where each example can be motivated by a biological application.

In Chap. 4 we have introduced the new notion of a story, which is a maximal acyclic subgraph of a directed graph in which only specified vertices can be sources or targets. We have proved some complexity results and designed some algorithms for enumerating all possible stories of a graph. From a theoretical point of view, the main question left open is to establish the complexity of the enumeration problem. Indeed the enumeration algorithm presented, even if it works well in practice, gives no guarantee on the delay between the output of two consecutive solutions. We address as a future work, exploiting the relationship between stories and subset feedback vertex sets that has been studied in [88] by applying *Measure and Conquer* approach [89]. From a practical point of view, for some graphs, the number of solutions found is extremely large and therefore the analysis of the results is compromised. Adding more constraints to the model could be a way to filter a priori the set of solutions.

A. Marino, *Analysis and Enumeration*, Atlantis Studies in Computing 6,
DOI 10.2991/978-94-6239-097-3_8

This observation on the size of the output leads us to consider the problem from a modelling point of view. For instance, the acyclicity constraint could be relaxed allowing cycles between white vertices. Moreover, the model could be enriched by exploring the information on the concentrations given by the metabolomics experiment. Notice that in this case the nature of the problem changes into an optimization problem. Another alternative is to consider integrated models, adding to the Metabolic network other layers of information such as regulation, or taking the stoichiometry of the reactions into account.

In Chap. 5 we showed that it is possible to enumerate all the bubbles, i.e. pairs of vertex disjoint paths, with a given source in a directed graph with linear delay. Moreover, it is possible to enumerate all bubbles, for all possible sources, in $O((|E|+|V|)(|C|+|V|))$ total time, where $|C|$ is the number of bubbles. This has required a non-trivial adaptation of Johnson's algorithm [10].

In Chap. 6 we showed the first optimal solution to list all the cycles of an undirected graph and all the paths from a given source to a given target. This result improves the Johnson's algorithm, that was still the theoretically most efficient in the case of undirected graphs. The main question arising from our work is whether it is possible to obtain an optimal algorithm to list all the paths and cycles in a directed graph in order to deal more efficiently with directed biological interaction networks, like gene regulatory networks, where the cycle enumeration have been discovered to be useful for several purposes. The main question arising from our work is whether it is possible to generalize our result, by finding a linear delay algorithm enumerating k-tuple of vertex disjoint paths.

Additionally, in Chap. 7 we have described and experimented new algorithms for enumerating all the diametral and radial vertices and computing the diameter and radius of directed and undirected (weighted) graphs: this enumeration problem is very particular since the number of solutions is polynomial in the size of the input.The growing interest towards centrality measures makes this problem interesting in the case of real-world networks in general, like social and web networks. In such a context, even if easy polynomial algorithms to find all the solutions exist, the huge size of real-world networks does not allow to run these existing algorithms. In the same scenario, even though our new algorithms have $O(nm)$ time complexity in the worst case, our experiments suggest that their execution for real-world networks requires time $O(m)$ in the case of diametral vertices and almost $O(m)$ in the case of the radial vertices. The main fundamental questions are now the followings. Why these algorithms, both in the directed and in the undirected version, are so effective in finding diameter, radius, and vertices with high and low eccentricity? Which one is the topological underlying property that can lead us to these results? Why real world graphs exhibit this property? Some progress has been done by [221], studying lower bound techniques for the diameter, but still a lot has to be done. Finally, it could be interesting to analyse a parallel implementation of the DiFUB algorithm. Indeed, the eccentricities of the vertices belonging to the same fringe set can be computed in parallel. Moreover, a variety of parallel BFS algorithms has been explored in the literature and can be integrated in the implementation of our algorithm.

References

1. Johnson, D. S., Papadimitriou, C. H., & Yannakakis, M. (1988). On generating all maximal independent sets. *Information Processing Letters*, *27*(3), 119–123.
2. Klein, C., Marino, A., & Sagot, M.-F. (2012). Paulo Vieira Milreu, and Matteo Brilli. *Briefings in Functional Genomics: Structural and Dynamical Analysis of Biological Networks*.
3. Madalinski, G., Godat, E., Alves, S., Lesage, D., Genin, E., Levi, P., et al. (2008). Direct introduction of biological samples into a ltq-orbitrap hybrid mass spectrometer as a tool for fast metabolome analysis. *Analytical Chemistry*, *80*(9), 3291–3303.
4. Acuña, V., Birmelé, E., Cottret, L., Crescenzi, P., Lacroix, V., Marchetti-Spaccamela, A., et al. (2011). Telling stories. In *Workshop on Graph Algorithms and Applications Selected For Submission to the Special Issue of Theoretical Computer Science in honor of Giorgio Ausiello in the Occasion of His 70th birthday, 2011*.
5. Acuña, V., Birmelé, E., Cottret, L., Crescenzi, P., Jourdan, F., Lacroix, V., et al. (October 2012). Telling stories: Enumerating maximal directed acyclic graphs with a constrained set of sources and targets. *Theoretical Computer Science*, *457*, 1–9.
6. Schwikowski, B., & Speckenmeyer, E. (2002). On enumerating all minimal solutions of feedback problems. *Discrete Applied Mathematics*, *117*(1–3), 253–265.
7. Acuña, V., Birmelé, E., Cottret, L., Crescenzi, P., Jourdan, F., Lacroix, V., et al. (2012). Metabolic stories: uncovering all possible scenarios for interpreting metabolomics data. In *First RECOMB Satellite Conference on Open Problems in Algorithmic Biology (RECOMB-AB)*.
8. Sacomoto, G., Kielbassa, J., Chikhi, R., Uricaru, R., Antoniou, P., Sagot, M. F., et al. (2012). KISSPLICE: De-novo calling alternative splicing events from RNA-seq data. *BMC Bioinformatics*, *13*(Suppl 6), S5.
9. Tiernan, J. C. (1970). An efficient search algorithm to find the elementary circuits of a graph. *Communications ACM*, *13*, 722–726.
10. Johnson, D. B. (1975). Finding all the elementary circuits of a directed graph. *SIAM Journal on Computing*, *4*(1), 77–84.
11. Birmelé, E., Crescenzi, P., Ferreira, R. A., Grossi, R., Lacroix, V., Marino, A., et al. (2012). Efficient bubble enumeration in directed graphs. In *String Processing and Information Retrieval—19th International Symposium, SPIRE* (pp. 118–129).
12. Klamt, S., & von Kamp, A. (2009). Computing paths and cycles in biological interaction graphs. *BMC Bioinformatics*, *10*(1), 181.
13. Cinquin, O., & Demongeot, J. (2002). Positive and negative feedback: Striking a balance between necessary antagonists. *Journal of Theoretical Biology*, *216*, 229–241.

A. Marino, *Analysis and Enumeration*, Atlantis Studies in Computing 6,
DOI 10.2991/978-94-6239-097-3

14. Thieffry, D. (2007). Dynamical roles of biological regulatory circuits. *Briefings in Bioinformatics*, *8*(4), 220–225.
15. Kwon, Y.-K., & Cho, K.-H. (2008). Coherent coupling of feedback loops: A design principle of cell signaling networks. *Bioinformatics*, *24*(17), 1926–1932.
16. Birmelé, E., Ferreira, R. A., Grossi, R., Marino, A., Pisanti, N., Rizzi, R., et al. (2013). Optimal listing of cycles and st-paths in undirected graphs. *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA* (pp. 1884–1896).
17. Mark, E. J. (2003). Newman. The structure and function of complex networks. *SIAM Review*, *45*, 167–256.
18. Crescenzi, P., Grossi, R., Lanzi, L., & Marino, A. (2012). On computing the diameter of real-world directed (weighted) graphs. In *Experimental Algorithms—11th International Symposium, SEA 2012* (pp. 99–110).
19. Crescenzi, P., Grossi, R., Imbrenda, C., Lanzi, L., & Marino, A. (2010). Finding the diameter in real-world graphs—experimentally turning a lower bound into an upper bound. In *Proceedings of the 18th Annual European Symposium on Algorithms—ESA 2010. Part I* (pp. 302–313).
20. Crescenzi, P., Grossi, R., Habib, M., Lanzi, L., & Marino, A. (2011). On computing the diameter of real-world undirected graphs. *Workshop on Graph Algorithms and Applications Selected for Submission to the Special Issue of Theoretical Computer Science in Honor of Giorgio Ausiello in the Occasion of His 70th Birthday, 2011*.
21. Crescenzi, P., Grossi, R., Habib, M., Lanzi, L., & Marino, A. (2013). On computing the diameter of real-world undirected graphs. *Theoretical Computer Science*, *514*, 84–95.
22. Backstrom, L., Boldi, P., Rosa, M., Ugander, J., & Vigna, S. (2011). *Four degrees of separation*. arXiv:1111.4570v1, 2011.
23. Backstrom, L., Boldi, P., Rosa, M., Ugander, J., & Vigna, S. (2012). Four degrees of separation. In *Web Science 2012, WebSci'12* (pp. 33–42).
24. Wasa, K. (2014). *Enumeration of enumeration algorithms*. http://www-ikn.ist.hokudai.ac.jp/wasa/enumeration_complexity.html.
25. Tan, P.-N., Steinbach, M., & Kumar, V. (2005). *Introduction to data mining* (1st ed.). Boston: Addison-Wesley Longman Publishing Co., Inc.
26. Uno, T. (2001). A fast algorithm for enumerating bipartite perfect matchings. In *Proceedings of the 12th International Symposium on Algorithms and Computation, ISAAC* (pp. 367–379).
27. Shioura, A., Tamura, A., & Uno, T. (1997). An optimal algorithm for scanning all spanning trees of undirected graphs. *SIAM Journal on Computing*, *26*(3), 678–692.
28. Tarjan, R. (1973). Enumeration of the elementary circuits of a directed graph. *SIAM Journal on Computing*, *2*(3), 211–216.
29. Read, R. C . & Tarjan, R. E. (1975). Bounds on backtrack algorithms for listing cycles, paths, and spanning trees. *Networks*, *5*(3), 237–252.
30. Eppstein, D. (1999). Finding the k shortest paths. *SIAM Journal on Computing*, *28*(2), 652–673.
31. Avis, D., & Fukuda, K. (1993). Reverse search for enumeration. *Discrete Applied Mathematics*, *65*, 21–46.
32. Uno, T. (2003). *Two general methods to reduce delay and change of enumeration algorithms*. NII Technical: Report.
33. Karp, R. M. (1972). Reducibility among combinatorial problems. In *complexity of computer computations* (pp. 85–103). New York: Plenum.
34. Makino, K., & Uno, T. (2004). New algorithms for enumerating all maximal cliques. In *Proceeding of the 9th Scandinavian Workshop on Algorithm Theory (SWAT 2004)* (pp. 260–272).
35. Kashiwabara, T., Masuda, S., Nakajima, K., & Fujisawa, T. (1992). Generation of maximum independent sets of a bipartite graph and maximum cliques of a circular-arc graph. *Journal of Algorithms*, *13*(1), 161–174.
36. Akkoyunlu, E. A. (1973). The enumeration of maximal cliques of large graphs. *SIAM Journal on Computing*, *2*(1), 1–6.

37. Tomita, E., Tanaka, A., & Takahashi, H. (2006). The worst-case time complexity for generating all maximal cliques and computational experiments. *Theoretical Computer Science*, *363*(1), 28–42.
38. Asai, T., Arimura, H., Uno, T., & Nakano, S.-I. (2003). Discovering frequent substructures in large unordered trees. In *6th International Conference, Discovery Science, DS* (pp. 47–61).
39. Yamanaka, K., Otachi, Y., & Nakano, S.-I. (2009). Efficient enumeration of ordered trees with kleaves (extended abstract). In *WALCOM: Algorithms and Computation, Third International Workshop, WALCOM* (pp. 141–150).
40. Uno, T., & Nakano, S.-I. (2003). *Efficient generation of rooted trees*. NII Technical: Report.
41. Nakano, S.-I., & Uno, T. (2004). Constant time generation of trees with specified diameter. In *Graph-Theoretic Concepts in Computer Science, 30th International Workshop, WG* (pp 33–45).
42. Nakano, S.-I., & Uno, T. (2005). Generating colored trees. In *Graph-Theoretic Concepts in Computer Science, 31st International Workshop, WG* (pp. 249–260).
43. Fukuda, K., & Matsui, T. (1989). Finding all the perfect matchings in bipartite graphs. *Applied Mathematics Letters*, *7*, 15–18.
44. Fukuda, K., & Matsui, T. (1992). Finding all minimum-cost perfect matchings in bipartite graphs. *Networks*, *22*(5), 461–468.
45. Uno, T. (1997). Algorithms for enumerating all perfect, maximum and maximal matchings in bipartite graphs. In *Proceedings of the 8th International Symposium on Algorithms and Computation, ISAAC'97* (pp. 92–101)
46. Chegireddy, C. R., & Hamacher, H. W. (1987). Algorithms for finding k-best perfect matchings. *Discrete Applied Mathematics*, *18*(2), 155–165.
47. Uno, T. (2001). A fast algorithm for enumerating non-bipartite maximal matchings. *Journal of National Institute of Informatics*, *3*, 89–97.
48. Lacroix, V., Cottret, L., Thébault, P., & Sagot, M.-F. (2008). An introduction to metabolic networks and their structural analysis. *Transactions on Computational Biology and Bioinformatics*, *5*(4), 594–617.
49. Cottret, L., & Jourdan, F. (2010). Graph methods for the investigation of metabolic networks in parasitology. *Parasitology*, *137*(9), 1393–1407.
50. Klamt, S., Haus, U.-U., & Theis, F. (2009). Hypergraphs and cellular networks. *PLoS Computational Biology5*(5), e1000385.
51. Lacroix, V., Cottret, L., Thébault, P., & Sagot, M.-F. (2008). An introduction to metabolic networks and their structural analysis. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, *5*(4), 594–617.
52. De Jong, H. (2002). Modeling and simulation of genetic regulatory systems: A literature review. *Journal of Computational Biology*, *9*, 67–103.
53. Klamt, S., Saez-Rodriguez, J., Lindquist, J. A., Simeoni, L., & Gilles, E. D. (2006). A methodology for the structural and functional analysis of signaling and regulatory networks. *BMC Bioinformatics*, *7*, 56.
54. Wang, R. S. & Albert, R. (2011). Elementary signaling modes predict the essentiality of signal transduction network components. *BMC Systems Biology*, *5*(1), 44.
55. Mardis, E. R. (2008). The impact of next-generation sequencing technology on genetics. *Trends in Genetics*, *24*(3), 133–141.
56. Steuer, R., Gross, T., Selbig, J., & Blasius, B. (2006). Structural kinetic modeling of metabolic networks. *Proceedings of the National Academy of Sciences*, *103*(32), 11868–11873.
57. Grimbs, S., Selbig, J., Bulik, S., Holzhütter, H.-G. G., & Steuer, R. (2007). The stability and robustness of metabolic states: identifying stabilizing sites in metabolic networks. *Molecular Systems Biology*, *3*, 146.
58. Steuer, R. (2007). Computational approaches to the topology, stability and dynamics of metabolic networks. *Phytochemistry*, *68*(16–18), 2139–2151.
59. Baldazzi, V., Ropers, D., Markowicz, Y., Kahn, D., Geiselmann, J., & de Jong H. (2010). The carbon assimilation network in escherichia coli is densely connected and largely sign-determined by directions of metabolic fluxes. *PLoS Computational Biology*, *6*(6).

60. Baldazzi, V., Ropers, D., Geiselmann, J., Kahn, D., & De Jong, H. (2012). Importance of metabolic coupling for the dynamics of gene expression following a diauxic shift in escherichia coli. *Journal of Theoretical Biology*, *295*, 100–115.
61. Kotte, O., Zaugg, J. B., & Heinemann, M. (2010). Bacterial adaptation through distributed sensing of metabolic fluxes. *Molecular Systems Biology*, *6*(1), 355.
62. Coulomb, S., Bauer, M., Bernard, D., & Marsolier-Kergoat, M.-C. (2005). Gene essentiality and the topology of protein interaction networks. *Proceedings of the Royal Society of London B*, *272*(1573), 1721–1725.
63. Costenbader, E., & Valente, T. W. (2003). The stability of centrality measures when networks are sampled. *Social Networks*, *25*(4), 283–307.
64. de Silva, E., Thorne, T., Ingram, P., Agrafioti, I., Swire, J., Wiuf, C., & Stumpf, M. P. (2006). The effects of incomplete protein interaction data on structural and evolutionary inferences. *BMC Biology*, *4*, 39.
65. Han, J.-D. J., Bertin, N., Hao, T., Goldberg, D. S., Berriz, G. F., Zhang, L. V., et al. (2004). Evidence for dynamically organized modularity in the yeast protein-protein interaction network. *Nature*, *430*(6995), 88–93.
66. Luscombe, N. M., Babu, M. M., Haiyuan, Y., Snyder, M., Teichmann, S. A., & Gerstein, M. (2004). Genomic analysis of regulatory network dynamics reveals large topological changes. *Nature*, *431*(7006), 308–312.
67. Konagurthu, A. S., & Lesk, A. M. (November 2008). Single and multiple input modules in regulatory networks. *Proteins*, *73*(2), 320–324.
68. Gopalacharyulu, P. V., Velagapudi, V. R., Lindfors, E., Halperin, E., & Orešič, M. (2009). Dynamic network topology changes in functional modules predict responses to oxidative stress in yeast. *Molecular Biosystems*, *5*(3), 276–287.
69. Ideker, T., & Krogan, N. J. (2012). Differential network biology. *Molecular Systems Biology*, *8*(1), 565.
70. Stumpf, M. P. H., Wiuf, C., & May, R. M. (2005). Subnets of scale-free networks are not scale-free: Sampling properties of networks. *Proceedings of the National Academy of Sciences of the United States of America*, *102*(12), 4221–4224.
71. Han, J.-D. D., Dupuy, D., Bertin, N., Cusick, M. E., & Vidal, M. (2005). Effect of sampling on topology predictions of protein-protein interaction networks. *Nature biotechnology*, *23*(7), 839–844.
72. Acuña, V., Milreu, P. V., Cottret, L., Marchetti-Spaccamela, A., Stougie, L. & Sagot, M.-F. (2012). Algorithms and complexity of enumerating minimal precursor sets in genome-wide metabolic networks. *Bioinformatics*, *28*(19), 2474–2483.
73. Wong, E., Baur, B., Quader, S., & Huang, C.-H. (2012). Biological network motif detection: Principles and practice. *Briefings in Bioinformatics*, *13*(2), 202–215.
74. Ciriello, G., & Guerra, C. (2008). A review on models and algorithms for motif discovery in protein-protein interaction networks. *Briefings in Functional Genomics and Proteomics*, *7*(2), 147–156.
75. Grochow, J. A., & Kellis, M. (2007). Network motif discovery using subgraph enumeration and symmetry breaking. In *Proceedings of the 11th International Conference on Research in Computational Molecular Biology (RECOMB)* (pp. 21–25). Springer.
76. Faust, K., Dupont, P., Callut, J., & van Helden, J. (2010). Pathway discovery in metabolic networks by subgraph extraction. *Bioinformatics*, *26*(9), 1211–1218.
77. Koyutürk, M., Grama, A., & Szpankowski, W. (2004). An efficient algorithm for detecting frequent subgraphs in biological networks. *Bioinformatics*, *20*(1), 200–207.
78. Zhang, S.-H., Ning, X.-M., & Zhang, X.-S. (2006). Identification of functional modules in a ppi network by clique percolation clustering. *Computers and Chemistry*, *30*(6), 445–451.
79. Georgii, E., Dietmann, S., Uno, T., Pagel, P., & Tsuda, K. (2009). Enumeration of condition-dependent dense modules in protein interaction networks. *Bioinformatics*, *25*(7), 933–940.
80. Eblen, J. D., Phillips, A., Rogers, G. L., & Langston, M. A. (2012). The maximum clique enumeration problem: Algorithms, applications, and implementations. *BMC Bioinformatics*, *13*(Suppl 10), S5.

81. Antonov, A. V., Dietmann, S., Wong, P., & Mewes, H. W. (2009). Ticl—a web tool for network-based interpretation of compound lists inferred by high-throughput metabolomics. *FEBS Journal*, *276*(7), 2084–2094.
82. Leader, D. P., Burgess, K., Creek, D., & Barrett, M. P. (2011). Pathos: A web facility that uses metabolic maps to display experimental changes in metabolites identified by mass spectrometry. *Rapid Communications in Mass Spectrometry*, *25*(22), 3422–3426.
83. Betzler, N. (2005). Steiner tree problems in the analysis of biological networks. *Ph.D. Dissertation Thesis*.
84. Milreu, P. V. (2012). Enumerating functional substructures of genome-scale metabolic networks: Stories, precursors and organisations. *Ph.D. Dissertation Thesis*.
85. Cottret, L., Wildridge, D., Vinson, F., Barrett, M. P., Charles, H., Sagot, M.-F., et al. (2010). Metexplore: A web server to link metabolomic experiments and genome-scale metabolic networks. *Nucleic Acids Research*, *38*(Web-Server-Issue), 132–137.
86. Garey, M. R., & Johnson, D. S. (1990). *Computers and intractability: A guide to the theory of NP-completeness*. New York: W. H. Freeman & Co.
87. Eiter, T., Makino, K., & Gottlob, G. (2008). Computational aspects of monotone dualization: A brief survey. *Discrete Applied Mathematics*, *156*(11), 2035–2049.
88. Fomin, F. V., Heggernes, P., Kratsch, D., Papadopoulos, C., & Villanger, Y. (2014). Enumerating minimal subset feedback vertex sets. *Algorithmica*, *69*(1), 216–231.
89. Fomin, F. V., Grandoni, F., & Kratsch, D. (2009). A measure and conquer approach for the analysis of exact algorithms. *Journal of the ACM*, *56*(5), 1–32.
90. Borassi, M., Crescenzi, P., Lacroix, V., Marino, R., Sagot, M.-F., & Milreu, P. V. (2013). Telling stories fast. In *Experimental Algorithms, 12th International Symposium, SEA* (pp. 200–211).
91. Peterlongo, P., Schnel, N., Pisanti, N., Sagot, M.-F., & Lacroix, V. (2010). Identifying snps without a reference genome by comparing raw reads. In *String Processing and Information Retrieval—17th International Symposium, SPIRE* (pp. 147–158).
92. Pevzner, P. A., Tang, H., & Tesler, G. (2004). De novo repeat classification and fragment assembly. In *Proceedings of the Eighth Annual International Conference on Computational Molecular Biology* (pp. 213–222).
93. Robertson, G., Schein, J., Chiu, R., Corbett, R., Field, M., Jackman, S. D., et al. (2010). De novo assembly and analysis of RNA-seq data. *Nature Methods*, *7*(11), 909–912.
94. Simpson, J. T., Wong, K., Jackman, S. D., Schein, J. E., Steven, J. M. J., & İnanç B. (2009). ABySS: A parallel assembler for short read sequence data. *Genome Research*, *19*(6), 1117–1123.
95. Zerbino, D. R., & Birney, E. (2008). Velvet: Algorithms for de novo short read assembly using de bruijn graphs. *Genome Research*, *18*(5), 821–829.
96. Iqbal, Z., Caccamo, M., Turner, I., Flicek, P., & McVean, G. (2012). De novo assembly and genotyping of variants using colored de bruijn graphs. *Nature Genetics*, *44*(2), 226–232.
97. Sammeth, M. (2009). Complete alternative splicing events are bubbles in splicing graphs. *Journal of Computational Biology*, *16*(8), 1117–1140.
98. Gusfield, D., Eddhu, S., & Langley, C. (2004). Optimal, efficient reconstruction of phylogenetic networks with constrained recombination. *Journal of Bioinformatics and Computational Biology*, *2*(1), 173–214.
99. Sacomoto, G., Lacroix, V., & Sagot, M.-F. (2013). A polynomial delay algorithm for the enumeration of bubbles with length constraints in directed graphs and its application to the detection of alternative splicing in rna-seq data. In *WABI* (pp. 99–111).
100. Maurya, M. R., Rengaswamy, R., & Venkatasubramanian, V. (2003). A systematic framework for the development and analysis of signed digraphs for chemical processes. 1. Algorithms and analysis. *Industrial and Engineering Chemistry Research*, *42*(20), 4789–4810.
101. Sontag, E. D. (2005). Molecular systems biology and control. *European Journal of Control*, *11*(4), 396–435.
102. Sussenguth, E. H. (1965). A graph-theoretic algorithm for matching chemical structures. *Journal of Chemical Documentation*, *5*(1), 36–43.

103. Welch, J. T., Jr. (1966). A mechanical analysis of the cyclic structure of undirected linear graphs. *Journal of the ACM (JACM)*, *13*(2), 205–210.
104. Bezem, G. J., & van Leeuwen, J. (1987). Enumeration in graphs. Technical Report RUU-CS-87-07, Utrecht University.
105. Mateti, P., & Deo, N. (1976). On algorithms for enumerating all circuits of a graph. *SIAM Journal on Computing*, *5*(1), 90–99.
106. Reinhard, D. (2005). *Graph theory (Graduate Texts in Mathematics)*. New York: Springer.
107. Syslo, M. M. (1981). An efficient cycle vector space algorithm for listing all cycles of a planar graph. *SIAM Journal on Computing*, *10*(4), 797–808.
108. Szwarcfiter, J. L., & Lauer, P. E. (1976). A search strategy for the elementary cycles of a directed graph. *BIT Numerical Mathematics*, *16*, 192–204.
109. Ponstein, J. (1966). Self-avoiding paths and the adjacency matrix of a graph. *SIAM Journal on Applied Mathematics*, *14*, 600–609.
110. Yau, S. S. (1967). Generation of all hamiltonian circuits, paths, and centers of a graph, and related problems. *IEEE Transactions on Circuit Theory*, *14*, 79–81.
111. Halford, T. R., & Chugg, K. M. (2004). Enumerating and counting cycles in bipartite graphs. In *IEEE Communication Theory Workshop* (Vol. 63).
112. Horváth, T., Gärtner, T., & Wrobel, S. (2004). Cyclic pattern kernels for predictive graph mining. In *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (pp. 158–167).
113. Liu H., & Wang J. (2006). A new way to enumerate cycles in graph. In *Telecommunications, 2006. AICT-ICIW'06. International Conference on Internet and Web Applications and Services/Advanced International Conference* (pp. 57–59).
114. Sankar, K., & Sarad, A. V. (2007). A time and memory efficient way to enumerate cycles in a graph. In *Intelligent and Advanced Systems* (pp. 498–500).
115. Wild, M. (2008). Generating all cycles, chordless cycles, and hamiltonian cycles with the principle of exclusion. *Journal of Discrete Algorithms*, *6*, 93–102.
116. Schott, R., & Staples, G. S. (2011). Complexity of counting cycles using zeons. *Computers and Mathematics with Applications*, *62*(4), 1828–1837.
117. Ferreira, R. A., Grossi, R., & Rizzi R. (2011). Output-sensitive listing of bounded-size trees in undirected graphs. In *19th Annual European Symposium on Algorithms—ESA* (pp. 275–286).
118. Tarjan, R. E. (1972). Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, *1*(2), 146–160.
119. Junker, B. H., & Schreiber, F. (2008). *Analysis of biological networks* (*Wiley Series in Bioinformatics*). Hoboken: Wiley-Interscience.
120. Leskovec, J., Lang, K. J., Dasgupta, A., & Mahoney, M. W. (2009). Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, *6*(1), 29–123.
121. Bansal, S., Khandelwal, S., & Meyers, L. (2009). Exploring biological network structure with clustered random networks. *BMC Bioinformatics*, *10*(1), 405.
122. Pržulj, N., Corneil, D. G., & Jurisica, I. (2006). Efficient estimation of graphlet frequency distributions in protein-protein interaction networks. *Bioinformatics*, *22*, 974–980.
123. Mislove, A., Marcon, M., Gummadi, P. K., Druschel, P., & Bhattacharjee, B. (2007). Measurement and analysis of online social networks. In *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement* (pp. 29–42).
124. Wilson, C., Boe, B., Sala, A., Puttaswamy, K. P. N., & Zhao, B. Y. (2009). User interactions in social networks and their implications. In *Proceedings of the 2009 EuroSys Conference* (pp. 205–218).
125. Wang, F., Moreno, Y., & Sun, Y. (2006). Structure of peer-to-peer social networks. *Physical Review E*, *73*, 036123.
126. Dong, Z.-B., Song, G.-J., Xie, K.-Q., & Wang J.-Y. (2009). An experimental study of large-scale mobile social network. In *Proceedings of the 18th International Conference on World Wide Web, WWW 2009* (pp. 1175–1176).

127. Mark, E. J. (2001). Newman. The structure of scientific collaboration networks. *Proceedings of the National Academy of Sciences of the United States of America*, *98*(2), 404–409.
128. Broder, A. Z., Kumar, R., Maghoul, F., Raghavan, P., Rajagopalan, S., Stata, R., et al. (2000). Graph structure in the web. *Computer Networks*, *33*(1–6), 309–320.
129. Kang, U., Tsourakakis, C. E., & Faloutsos, C. (2011). PEGASUS: Mining peta-scale graphs. *Knowledge and Information Systems*, *27*(2), 303–325.
130. Zwick, U. (2000). All pairs shortest paths using bridging sets and rectangular matrix multiplication. *Journal of the ACM*, *49*, 2002.
131. Palmer, C. R., Gibbons, P. B., & Faloutsos, C. (2002). ANF: A fast and scalable tool for data mining in massive graphs. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (pp. 81–90).
132. Boldi, P., Rosa, M., & Vigna, S. (2011). Hyperanf: approximating the neighbourhood function of very large graphs on a budget. In *Proceedings of the 20th International Conference on World Wide Web, WWW 2011* (pp. 625–634).
133. Kang, U., Tsourakakis, C. E. (2011). Ana paula appel, christos faloutsos, and jure leskovec. Hadi: Mining radii of large graphs. *TKDD*, *5*(2), 8.
134. Cohen, E. (1994). Estimating the size of the transitive closure in linear time. In *35th Annual Symposium on Foundations of Computer Science* (pp. 190–200).
135. Cohen, E., & Kaplan, H. (2007). Summarizing data using bottom-k sketches. *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing, PODC, 2007* (pp. 225–234).
136. Cohen, E. (1997). Size-estimation framework with applications to transitive closure and reachability. *Journal of Computer and System Sciences*, *55*(3), 441–453.
137. Cohen, E., & Kaplan, H. (2008). Tighter estimation using bottom k sketches. *PVLDB*, *1*(1), 213–224.
138. Cohen, E., & Kaplan, H. (2007). Bottom-k sketches: Better and more efficient estimation of aggregates. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS 2007* (pp. 353–354).
139. Leskovec, J., & Faloutsos, C. (2006). Sampling from large graphs. In *Proceedings of the Twelfth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (pp. 631–636).
140. Latapy, M., & Magnien, C. (2006) Measuring fundamental properties of real-world complex networks. *CoRR, abs/cs/0609115*.
141. Takes, F. W. & Kosters, W. A. (2011). Determining the diameter of small world networks. In *Proceedings of the 20th ACM Conference on Information and Knowledge Management, CIKM 2011* (pp. 1191–1196).
142. Boldi, P., & Vigna, S. (2004). The webgraph framework I: Compression techniques. In *Proceedings of the 13th international conference on World Wide Web, WWW 2004* (pp. 595–602).
143. Crescenzi, P., Grossi, R., Lanzi, L., & Marino, A. (2011). A comparison of three algorithms for approximating the distance distribution in real-world graphs. In *Theory and Practice of Algorithms in (Computer) Systems—First International ICST Conference, TAPAS 2011* (pp. 92–103).
144. Junker, B. H., Koschützki, D., & Schreiber, F. (2006). Exploration of biological network centralities with centibin. *BMC Bioinformatics*, *7*, 219.
145. Gräßler, J., Koschützki, D., & Schreiber, F. (2012). Centilib: comprehensive analysis and exploration of network centralities. *Bioinformatics*, *28*(8), 1178–1179.
146. Koschützki, D., & Schreiber, F. (2008). Centrality analysis methods for biological networks and their application to gene regulatory networks. *Gene Regulation and Systems Biology*, *2*, 193–201.
147. Pavlopoulos, G., Secrier, M., Moschopoulos, C., Soldatos, T., Kossida, S., Aerts, J., et al. (2011). Using graph theory to analyze biological networks. *BioData Mining*, *4*(1), 10.
148. Mason, O., & Verwoerd, M. (2007). Graph theory and networks in biology. *Systems Biology, IET*, *1*(2), 89–119.

149. Scardoni, G., & Laudanna, C. (2012). Centralities based analysis of complex networks. In Y. Zhang (Ed.), *New frontiers in graph theory*. InTech, ISBN: 978-953-51-0115-4. doi:10.5772/35846. Available from: http://www.intechopen.com/books/new-frontiers-in-graph-theory/centralities-based-analysis-of-networks
150. Hu, Z., Hung, J.-H., Wang, Y., Chang, Y.-C., Huang, C.-L., Huyck M., et al. (2009). Visant 3.5: multi-scale network visualization, analysis and inference based on the gene ontology. *Nucleic Acids Research*, *37*, W115–W121.
151. Baur, M., Benkert, M., Brandes, U., Cornelsen, S., Gaertler, M., Köpf, B., et al. (2001). Visone. In *Graph Drawing* (pp. 463–464)
152. Batagelj, V., & Mrvar, A. (1998). Pajek-program for large network analysis. *Connections*, *21*(2), 47–57.
153. Gräßler, J., Koschützki, D., & Schreiber, F. (2012). Centilib: comprehensive analysis and exploration of network centralities. *Bioinformatics*, *28*(8), 1178–1179.
154. Hawoong, J., Mason, S. P., Barabási, A.-L., & Oltvai, Z. N. (2001). Lethality and centrality in protein networks. *Nature*, *411*(6833), 41–42.
155. He, X., & Zhang, J. (2006). Why do hubs tend to be essential in protein networks? *PLoS Genet*, *2*(6), e88.
156. Albert, R., Jeong, H., & Barabasi, A.-L. (2000). Error and attack tolerance of complex networks. *Nature*, *406*(6794), 378–382.
157. Wuchty, S., & Almaas, E. (2005). Peeling the yeast protein network. *Proteomics*, *5*(2), 444–449.
158. Wuchty, S. (2002). Interaction and domain networks of yeast. *Proteomics*, *2*(12), 1715–1723.
159. Zotenko, E., Mestre, J., O'Leary, D. P., & Przytycka, T. M. (2008). Why do hubs in the yeast protein interaction network tend to be essential: reexamining the connection between the network topology and essentiality. *PLoS Computational Biology*, *4*(8), e1000140.
160. Batada, N. N., Reguly, T., Breitkreutz, A., Boucher, L., Breitkreutz, B.-J., Hurst, L. D., et al. (2006). Stratus not altocumulus: a new view of the yeast protein interaction network. *PLoS Biology*, *4*(10), e317.
161. Reguly, T., Breitkreutz, A., Boucher, L., Breitkreutz, B.-J. J., Hon, G. C., Myers, C. L. et al. (2006). Comprehensive curation and analysis of global interaction networks in Saccharomyces cerevisiae. *Journal of Biology*, *5*, 11.
162. Ekman, D., Light, S., Bjorklund, A., & Elofsson, A. (2006). What properties characterize the hub proteins of the protein-protein interaction network of Saccharomyces cerevisiae? *Genome Biology*, *7*, R45.
163. Aragues, R., Sali, A., Bonet, J., Marti-Renom, M. A., & Oliva, B. (2007). Characterization of protein hubs by inferring interacting motifs from protein interactions. *PLoS Computational Biology*, *3*(9), 1761–1771.
164. Barabasi, A.-L., & Oltvai, Z. N. (2004). Network biology: Understanding the cell's functional organization. *Nature Reviews Genetics*, *5*(2), 101–113.
165. Lima-Mendez, G., & van Helden, J. (2009). The powerful law of the power law and other myths in network biology. *Molecular BioSystems*, *5*(12), 1482–1493.
166. Mark, E. J. (2002). Newman. Assortative mixing in networks. *Physical Review Letters*, *89*(20), 208701.
167. Maslov, S., & Sneppen, K. (2002). Specificity and stability in topology of protein networks. *Science*, *296*(5569), 910–913.
168. Park, Juyong, & Barabási, Albert-László. (2007). Distribution of node characteristics in complex networks. *Proceedings of the National Academy of Sciences*, *104*(46), 17916–17920.
169. Jiang, X., Liu, B., Jiang, J., Zhao, H., Fan, M., Zhang, J., et al. (2008). Modularity in the genetic disease-phenotype network. *FEBS Letters*, *582*(17), 2549–2554.
170. Nacher, J., & Araki, N. (2011). On the relation between structure and biological function in transcriptional networks and ncRNA-mediated interactions. In *2011 International Conference on Bioscience, Biochemistry and Bioinformatics IPCBEE* (Vol. 5, pp. 348–352).
171. Latora, V., & Marchiori, M. (2007). A measure of centrality based on network efficiency. *New Journal of Physics*, *9*(6), 188.

172. Freeman, L. C. (1977). A set of measures of centrality based on betweenness. *Sociometry*, *40*(1), 35–41.
173. Ng, S.-K., & Li, X.-L. (2009). *Biological data mining in protein interaction networks*. Hershey: Information Science Reference—Imprint of: IGI Publishing.
174. Hsu, C.-L., Huang, Y.-H., Hsu, C.-T., & Yang, U.-C. (2011). Prioritizing disease candidate genes by a gene interconnectedness-based approach. *BMC Genomics*, *12*(Suppl 3), S25.
175. Eppstein, D., & Wang, J. (2001). Fast approximation of centrality. In *Proceedings of the Twelfth Annual Symposium on Discrete Algorithms* (pp. 228–229).
176. Wuchty, S., & Stadler, P. F. (2003). Centers of complex networks. *Journal of Theoretical Biology*, *223*(1), 45–53.
177. Chavali, S., Barrenas, F., Kanduri, K., & Benson, M. (2010). Network properties of human disease genes with pleiotropic effects. *BMC Systems Biology*, *4*(1), 78.
178. Yu, H., Kim, P. M., Sprecher, E., Trifonov, V., & Gerstein, M. (2007). The Importance of Bottlenecks in Protein Networks: Correlation with Gene Essentiality and Expression Dynamics. *PLoS Computational Biology*, *3*(4), e59.
179. McDermott, J. E., Taylor, R. C., Yoon, H., & Heffron, F. (2009). Bottlenecks and hubs in inferred networks are important for virulence in Salmonella typhimurium. *Journal of computational biology : A journal of computational molecular cell biology*, *16*(2), 169–180.
180. Caretta-Cartozo, C., De Los Rios, P., Piazza, F., & Liò, P. (2007). Bottleneck genes and community structure in the cell cycle network of S. pombe. *PLoS Computational Biology*, *3*(6), e103.
181. Vallabhajosyula, R. R. & Raval, A. (2010). Computational modeling in systems biology. In *Systems biology in drug discovery and development volume 662 of methods in molecular biology, chapter 5* (pp. 97–120). Totawa: Humana Press.
182. Chavali, A. K., Blazier, A. S., Tlaxca, J. L., Jensen, P. A., Pearson, R. D., & Papin, J. A. (2012). Metabolic network analysis predicts efficacy of FDA-approved drugs targeting the causative agent of a neglected tropical disease. *BMC Systems Biology*, *6*, 27.
183. Chen, L. C., Yeh, H. Y., Yeh, C. Y., Arias, C., & Soo, V. W. (2012). Identifying co-targets to fight drug resistance based on a random walk model. *BMC Systems Biology*, *6*(1), 5.
184. Rahman, S. A., & Schomburg, D. (2006). Observing local and global properties of metabolic pathways: 'Load points' and 'choke points' in the metabolic networks. *Bioninformatics*, *22*(14), 1767–1774.
185. Smith, T. G., & Hoover, T. R. (2009). Deciphering bacterial flagellar gene regulatory networks in the genomic era. *Advances in Applied Microbiology*, *67*, 257–295.
186. Newman, M. E. J. (2005). A measure of betweenness centrality based on random walks. *Social Networks*, *27*(1), 39–54.
187. Bonacich, P. (2007). Some unique properties of eigenvector centrality. *Social Networks*, *29*(4), 555–564.
188. Perra, N., & Fortunato, S. (2008). Spectral centrality measures in complex networks. *Physical Review E (Statistical, Nonlinear, and Soft Matter Physics)*, *78*(3), 036107.
189. Ding, D.-W. & He, X.-Q. (2010) Application of eigenvector centrality in metabolic networks. In *Proceedings of the 2nd International Conference on Computer Engineering and Technology* (pp. V1-89–V1-91).
190. Estrada, E. (2006). Virtual identification of essential proteins within the protein interaction network of yeast. *Proteomics*, *6*(1), 35–40.
191. Estrada, E., & Rodriguez-Velazquez, J. A. (2005). Subgraph centrality in complex networks. *Physical Review E*, *71*(5), 056103.
192. Li, M., Zhang, H., Wang, J., & Pan, Y. (2012). A new essential protein discovery method based on the integration of protein-protein interaction and gene expression data. *BMC Systems Biology*, *6*(1), 15.
193. Wang, J., Li, M., Wang, H., & Pan, Y. (2012). Identification of essential proteins based on edge clustering coefficient. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, *9*(4), 1070–1080.

194. Lanzi, L. (2012). Complex networks: Algorithms, analysis, and models. *Ph.D. Dissertation Thesis*.
195. Magnien, C., Latapy, M., & Habib, M. (2009). Fast computation of empirically tight bounds for the diameter of massive graphs. *Journal of Experimental Algorithmics*, *13*, 10. http://dl.acm.org/citation.cfm?doid=1412228.1455266.
196. Dijkstra, E. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, *1*, 269–271.
197. Research Group on Graph Theory and Department of the Universitat Politècnica de Catalunya (UPC) Combinatorics. (2010). *The degree diameter problem for general graphs*. Retrieved from http://www-mat.upc.es/grup_de_grafs/.
198. SNAP. (2009). *Stanford Network Analysis Package (SNAP)*. Website http://snap.stanford.edu.
199. WebGraph. (2001). WebGraph http://webgraph.di.unimi.it/.
200. Christian Sommer. (2009). *Christian Sommer's homepage*. http://www.sommer.jp/graphs/.
201. HPRD. (2003). *Human potein reference database*. http://www.hprd.org/.
202. The Jena Protein-Protein Interaction Website. (2009). http://ppi.fli-leibniz.de/.
203. iPfam. (2009). *iPfam: The Protein Domain Interactions Database*. http://ipfam.sanger.ac.uk/.
204. Wu, J., Vallenius, T., Ovaska, K., Westermarck, J., Mäkelä, T. P., & Hautaniemi, S. (2009). Integrated network analysis platform for protein-protein interactions. *Nature Methods*, *6*, 75–77.
205. Crescenzi, P., Lanzi, L., & Marino, A. (2012). *Lasagne: Laboratory of algorithms, models, and analysis of graphs and networks*. http://amici.dsi.unifi.it/lasagne/.
206. Integrated protein-protein interaction database of Synechocystis sp. (2007). PC 6803. http://bioportal.kobic.re.kr/SynechoNET/.
207. Pajek Dataset. (2006). http://vlado.fmf.uni-lj.si/pub/networks/data/default.htm.
208. TrustLet. (2007). TrustLet Website. http://www.trustlet.org.
209. Clusters and Communities, Overlapping Dense Groups in Networks. (2005). http://hal.elte.hu/cfinder/.
210. Arenas, A., & Duch, J. (2005). Community identification using extremal optimization. *Physical Review E*, *72*, 027104.
211. tnet. tnet package, analysis of weighted, two-mode, and longitudinal networks. http://opsahl.co.uk/tnet/datasets/.
212. Palla, G., Farkas, I. J., Pollner, P., Derényi, I., & Vicsek T. (2008). Fundamental statistical features and self-similar properties of tagged networks. *New Journal of Physics*, *10*(12), 20.
213. Norlen, K., Lucas, G., Gebbie, M., & Chuang, J. (2002). EVA: Extraction, visualization and analysis of the telecommunications and media ownership network. In *Proceedings of International Telecommunications Society 14th Biennial Conference (ITS2002)*. Seoul Korea: International Telecommunications Society.
214. Gleiser, Pablo, & Danon, Leon. (2003). Community structure in Jazz. *Advances in Complex Systems*, *6*(4), 565–573.
215. Boguna, M., Pastor-Satorras, R., Díaz-Guilera, A., & Arenas, A. (2004). Models of social networks based on social distance attachment. *Physical Review E*, *70*(5), 343.
216. Sandbox. (2010). Webscope from Yahoo! Labs. Data available, as explained at http://sandbox.yahoo.com/.
217. Zhao, B. Y. (2010). CURRENT LAB: Social networking project. Data available, as explained at http://current.cs.ucsb.edu/facebook/.
218. Aron, U. (2002). *Uri Alon Lab*. http://www.weizmann.ac.il/mcb/UriAlon/.
219. Ajwani, D., Meyer, U., & Veith, D. (2012). I/o-efficient hierarchical diameter approximation. In *Algorithms—ESA 2012–20th Annual European Symposium* (pp. 72–83).
220. Ajwani, D., Beckmann, A., Meyer, U., & Veith, D. (2012). I/o-efficient approximation of graph diameters by parallel cluster growing—a first experimental study. In *ARCS 2012 Workshops* (pp. 493–504).
221. Chepoi, V., Dragan, F. F., Estellon, B., Habib, M., & Vaxès, Y. (2008). Diameters, centers, and approximating trees of delta-hyperbolicgeodesic spaces and graphs. In *Proceedings of the 24th ACM Symposium on Computational Geometry* (pp. 59–68).

222. Borassi, M., Crescenzi, P., Habib, M., Kosters, W. A., Marino, A., & Takes, F. W. (2011). On the solvability of the six degrees of kevin bacon game—a faster graph diameter and radius computation method. In: *Fun with Algorithms—7th International Conference, FUN* (pp. 52–63).